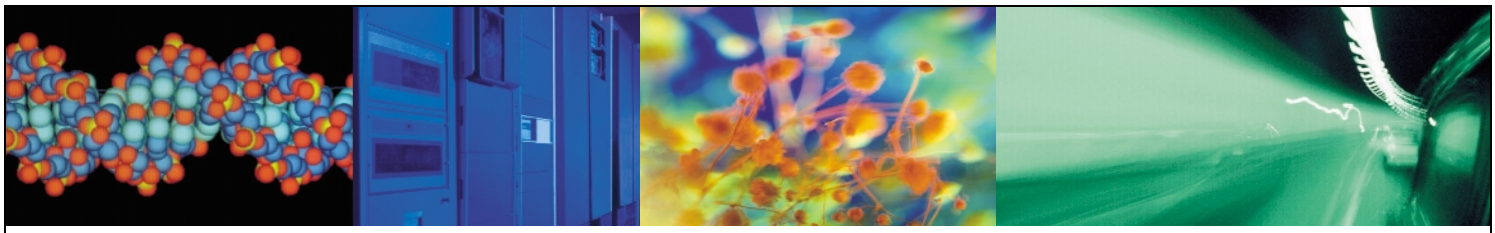

PathScale™

PATHSCALE EKO COMPILER SUITE™

USER GUIDE

PathScale EKO Compiler Suite
User Guide

Release 1.2

PathScale, Inc.

Copyright © 2004 PathScale, Inc. All Rights Reserved.

PathScale, the PathScale EKO Compiler Suite, and Accelerating Cluster Performance are trademarks of PathScale, Inc. All other trademarks belong to their respective owners.

In accordance with the terms of their valid PathScale customer agreements, customers are permitted to make electronic and paper copies of this document for their own exclusive use.

All other forms of reproduction, redistribution, or modification is prohibited without the prior express written permission of PathScale, Inc.

Document number: 1-02404-03
Last generated on June 28, 2004

Contents

1 Introduction **1**

1.1 Conventions used in this document 2

1.2 Other resources 2

2 Compiler Quick Reference **5**

2.1 What you installed 5

2.2 How to invoke the PathScale EKO compilers 6

2.3 Input file types 7

2.4 Other input files 8

2.5 Common compiler options 9

2.6 Shared libraries 9

2.7 Large file support 10

2.8 Large object support 10

2.8.1 Support for "large" memory model 11

2.9 Debugging 11

2.10 Profiling: Locate your program's hot spots 12

2.11 Taskset: Assigning a process to a specific CPU 13

3 The PathScale EKO Fortran compiler 15

3.1 Using the Fortran compiler . . . 15

3.1.1 Fixed-form and free-form files . . . 16

3.2 Modules . . . 17

3.3 Extensions . . . 17

3.3.1 Promotion of REAL and INTEGER types . . . 18

3.3.2 Cray pointers . . . 18

3.3.3 Directives . . . 18

3.4 Compiler and runtime features . . . 19

3.4.1 Preprocessing source files . . . 19

3.4.2 Explain . . . 19

3.4.3 Mixed code . . . 20

3.4.4 Bounds checking . . . 20

3.4.5 Pseudo-random numbers . . . 20

3.5 Runtime I/O compatibility . . . 21

3.5.1 Performing endian conversions . . . 21

3.5.1.1 The assign command . . . 21

3.5.1.2 Using the wildcard option . . . 21

3.5.1.3 Converting data and record headers . . . 22

3.5.1.4 The ASSIGN() procedure . . . 22

3.6 Source code compatibility . . . 22

3.6.1 Fortran KINDs . . . 22

3.6.2 Fortran 95 . . . 23

3.7 Library compatibility . . . 23

3.7.1 Name mangling . . . 24

3.7.2 ABI compatibility . . . 24

3.7.3 Linking with g77-compiled libraries 25

3.7.3.1 AMD Core Math Library (ACML) 26

3.8 Debugging and troubleshooting 26

3.8.1 Writing to constants can cause crashes 27

3.8.2 Aliasing: -OPT:alias=no_parm 27

4 The PathScale EKO C/C++ compiler 29

4.1 Using the C/C++ compilers 30

4.2 Compiler and runtime features 31

4.2.1 Preprocessing source files 31

4.2.2 Mixing code 31

4.2.3 Linking 31

4.3 Debugging 31

4.4 GCC extensions not supported 32

5 Porting and compatibility 33

5.1 Getting started 33

5.2 Cookbook 34

5.3 Compatibility 34

5.3.1 GCC compatibility wrapper script 34

5.3.2 Modifying existing scripts 35

5.3.3 Name mangling 35

5.4 Compiler options for porting and correctness 36

5.5 Fortran compiler stack size 36

6 Tuning Quick Reference **39**

6.1 Basic optimization . . . 39

6.2 IPA . . . 39

6.3 Feedback directed optimization . . . 40

6.4 Aggressive optimization . . . 40

6.5 Performance analysis . . . 41

6.6 Optimize your hardware . . . 41

7 Tuning options **43**

7.1 Basic optimizations: The `-O` flag . . . 43

7.2 Syntax for complex optimizations (`-CG, -IPA, -LNO -OPT, -WOPT`) . . 45

7.3 Inter-Procedural Analysis (IPA) . . . 45

7.3.1 Size and correctness limitations to IPA . . . 47

7.4 Loop Nest Optimization (LNO) . . . 47

7.4.1 Loop fusion and fission . . . 48

7.4.2 Cache size specification . . . 48

7.4.3 Cache blocking, loop unrolling, interchange transformations . . . 49

7.4.4 Prefetch . . . 49

7.4.5 Vectorization . . . 50

7.5 Code Generation (-CG:) . . . 50

7.6 Feedback Directed Optimization (FDO) . . . 51

7.7 Aggressive optimizations . . . 52

7.7.1 Alias analysis . . . 52

7.7.2 Numerically unsafe optimizations . . . 53

7.7.3 IEEE 754 compliance . . . 54

7.7.3.1 Arithmetic . . . 54

7.7.3.2 Roundoff 55
7.7.4 Other unsafe optimizations 56
7.7.5 Assumptions about numerical accuracy 56
7.8 Opteron performance 57
7.8.1 Hardware setup 57
7.8.2 BIOS setup 58
7.8.3 Multiprocessor memory 58
7.8.4 Kernel and system effects 58
7.8.5 Tools and APIs 59
7.8.6 Testing memory latency and bandwidth 59

8 Examples **61**
8.1 Compiler flag tuning and profiling with `pathprof` 61

9 Debugging and troubleshooting **65**
9.1 Subscription Manager problems 65
9.2 Debugging 65
9.3 Large object support 65
9.4 Using `-ipa` and `-Ofast` 66
9.5 Tuning 67

A Environment variables **69**
A.1 Environment variables for use with C 69
A.2 Environment variables for use with C++ 69
A.3 Environment variables for use with Fortran 69
A.4 Language independent environment variables 70

B Supported intrinsics **71**

C Glossary **79**

Chapter 1

Introduction

This User Guide covers how to use the PathScale EKO Compiler Suite™ compilers; how to configure them, how to use them to optimize your code, and how to get the best performance from them. This guide also covers the language extensions and differences from the other commonly available language compilers.

The PathScale EKO Compiler Suite now generates both 32-bit and 64-bit code. 64-bit code is the default; to generate 32-bit code use `-m32` on the command line. See the `eko` man page for details.

The information in this guide is organized into these sections:

- Chapter 2 is a quick reference to using the PathScale EKO compilers
- Chapter 3 covers the PathScale EKO Fortran compiler
- Chapter 4 covers the PathScale EKO C/C++ compilers
- Chapter 5 provides suggestions for porting and compatibility
- Chapter 6 is a Tuning Quick Reference, with tips for getting faster code
- Chapter 7 discusses tuning options in more detail
- Chapter 8 provides examples of optimizing code
- Chapter 9 covers debugging and troubleshooting code
- Appendix A lists environmental variables used with the compilers
- Appendix B is a list of the supported intrinsics
- Appendix C provides descriptions of the optimization flags
- Appendix D is a glossary of terms associated with the compilers

1.1 Conventions used in this document

These conventions are used throughout the PathScale documentation.

Convention	Meaning
`command`	Fixed-space font is used for literal items such as commands, files, routines, and pathnames.
variable	Italic typeface is used for variable names or concepts being defined.
`user input`	Bold, fixed-space font is used for literal items the user types in. Output is shown in non-bold, fixed-space font.
$	Indicates a command line prompt
#	Command line prompt as root
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.
NOTE:	Indicates important information

1.2 Other resources

The PathScale EKO Compiler Suite product documentation set includes:

- *The PathScale EKO Compiler Suite Install Guide*
- *The PathScale EKO Compiler Suite User Guide*
- *The PathScale EKO Compiler Suite Support Guide*

There are also online manual pages (“man pages”) available describing the flags and options for the PathScale EKO Compiler Suite. You can type "`man -k pathscale`" or "`apropos pathscale`" to get a list of all the PathScale man pages on your system. This feature does not work on SLES 8.

Please see the PathScale website at `http://www.pathscale.com/support.html` for further information about current releases and developer support. In addition, you may want to refer to these books for more information on high performance computing, compilers, and language usage.

- *Fortran 95 Explained* by Metcalf, M. and Reid, J., Oxford University Press, 1996. ISBN 0-19-851888-8
- *C Programming Language* by Brian W. Kernighan, Dennis Ritchie, Dennis M. Ritchie, Prentice Hall, 1988, 2nd edition, ISBN 0-13-110362-8

- *The C++ Programming Language* by Bjarne Stroustrup, Addison-Wesley Publishing Company, 2000, 3rd edition, ISBN 0-20-170073-5
- *The Practice of Programming* by Brian W. Kernighan and Rob Pike, Addison-Wesley Publishing Company, 1st edition, 1999, ISBN 0-20-161586-X
- *High Performance Computing* by Kevin Doud, O'Reilly & Associates, Inc., 1993. ISBN 1-56592-032-5

Chapter 2

Compiler Quick Reference

This chapter describes how to get started using the PathScale EKO Compiler Suite. The compilers follow the standard conventions of Unix and Linux compilers. They produce code that follows the Linux/AMD64 ABI and run on the AMD64 family of chips. This means that object files produced by the PathScale EKO compilers can link with object files produced by other Linux/AMD64-compliant compilers such as Red Hat and SuSE GNU `gcc`, `g++`, and `g77`. AMD64 is AMD's 64-bit extension to Intel's IA32 architecture, often referred to as "x86".

2.1 What you installed

The PathScale EKO Compiler Suite™ includes optimizing compilers and runtime support for C, C++, and Fortran.

Depending on the type of subscription you purchased, you enabled some or all of the following:

- PathScale EKO C Compiler for AMD64 architecture
- PathScale EKO C++ Compiler for AMD64 architecture
- PathScale EKO Fortran Compiler for AMD64 architecture
- Documentation
- Libraries
- Subscription Manager client[1]
- Subscription Manager server (optional)[2]

[1] You must have a valid subscription (and associated subscription file) in order to run the compiler.
[2] The PathScale Subscription Manager server is required for floating subscriptions.

- GNU binutils

For more details on installing the PathScale EKO compilers, see the *PathScale EKO Compiler Suite Install Guide*.

2.2 How to invoke the PathScale EKO compilers

The PathScale EKO Compiler Suite has three different front-ends to handle programs written in C, C++, and Fortran, and it has common optimization and code generation components that interface with all the language front-ends. The language your program uses determines which command (driver) name to use:

Language	Command Name	Compiler Name
C	`pathcc`	PathScale EKO C compiler
C++	`pathCC`	PathScale EKO C++ compiler
Fortran 77 Fortran 90 Fortran 95	`pathf90`	PathScale EKO Fortran compiler

There are online manual pages (“man pages”) with descriptions of the large number of command line options that are available. You can type "`man -k pathscale`" or "`apropos pathscale`" to get a list of all the PathScale man pages on your system. To view the general man page for the compilers, type `man pathscale_intro` at the command line.

If invoked with the flag `-v`, the compilers will emit some text that identifies the version. For example:

```
$ pathcc -v
PathScale Compiler Suite(TM): Version 1.2
gcc version 3.3.1 (PathScale 1.2 driver)
```

You can create a common example program called `world.c`:

```
#include <stdio.h>
main() {
 printf ("Hello World!\n");
}
```

Then you can compile it from your shell prompt very simply:

```
# pathcc world.c
```

The default output file for the pathcc-generated executable is named `a.out`. You can execute it and see the output:

```
# ./a.out
Hello World!
```

As with most compilers, you can use the `-o <filename>` option to give your program executable file the desired name.

NOTE: By default the PathScale EKO compilers generate 64-bit code. To generate 32-bit code you must specify `-m32` on the command line when you compile. See the `eko` man pages for details.

2.3 Input file types

The name for a source file usually has the form `filename.ext`, where `ext` is a one to three character extension used on a source code file that can have various meanings:

Extension	Implication to the driver
`.c`	C source file that will be preprocessed

`.C`
`.cc`
`.cpp`
`.cxx`
	C++ source file that will be preprocessed

`.f` `.f90`	Fortran source file `.f` is fixed format, no preprocessor `.f90` is freeform format, no preprocessor
`.F` `.F90`	Fortran source file `.F` is fixed format, invokes preprocessor `.F90` is freeform format, invokes preprocessor

For Fortran files with the extensions `.f` or `.f90`, you can use the `-ftpp` or `-cpp` options on the `pathf90` command line to invoke the C preprocessor. See Section 3.4.1 for more information on preprocessing.

The compiler drivers can use the extension to determine which language front-end to invoke. For example, some mixed language programs can be compiled with a single command:

```
# pathf90 stream_d.f second_wall.c -o stream
```

The `pathf90` driver will use the `.c` extension to know that it should automatically invoke the C front-end on the `second_wall.c` module and link the generated object files into the `stream` executable.

NOTE: GNU make does not contain a rule for generating object files from Fortran `.f90` files. You can add the following rules to your project Makefiles to achieve this:

```
$.o:   %.f90
       $(FC) $(FFLAGS) -c $<
$.o:   %.F90
       $(FC) $(FFLAGS) -c $<
```

You may need to modify this for your project, but in general it should follow this form.

For more information on compatibility and porting existing code, see Section 5. Information on GCC compatibility and a wrapper script you can use for your build packages can be found in Section 5.3.1.

2.4 Other input files

Other possible input files, common to both C/C++ and Fortran, are assembly-language files, object files, and libraries as inputs on the command line.

Extension	Implication to the driver
`.i`	Preprocessed source file
`.s`	Assembly language file
`.o`	object file
`.a`	a static library of object files
`.so`	a library of shared (dynamic) object files

2.5 Common compiler options

The PathScale EKO Compiler Suite has command line options that are similar to many other Linux or Unix compilers:

Option	What it does
`-c`	generates an intermediate object file for each source file, but doesn't link
`-g`	produces debugging information to allow full symbolic debugging
`-I<dir>`	Add <*path*> to the directories searched by preprocessor for include file resolution.
`-l<library>`	Searches the library specified during the linking phase for unresolved symbols.
`-L<dir>`	Add <path> to the directories searched during the linking phase for libraries.
`-lm`	links using the `libm` math library. This is typically required in C programs that use functions such as `exp()`, `log()`, `sin()`, `cos()`.
`-o <filename>`	generates the named executable (binary) file
`-O3`	generates a highly optimized executable, generally numerically safe
`-O or -O2`	generates an optimized executable that is numerically safe. (This is also the default if no `-O` flag is used.)
`-pg`	generates profile information suitable for the analysis program `pathprof`

Many more options are available and described in the man pages (`pathscale_intro, pathcc`, `pathf90`, `pathCC`, `eko`) and Chapter 7 in this document.

2.6 Shared libraries

The PathScale EKO Compiler Suite includes shared versions of the runtime libraries that the compilers use. The shared libraries are packaged in the `pathscale-compilers-libs` package. The compiler will use these shared libraries by default when linking executables and shared objects. As a result, if you link a program with these shared libraries, you must install them on systems where that program will run.

You should continue to use the static versions of the runtime libraries if you wish to obtain maximum portability or peak performance. The latter is the case because the compiler cannot optimize shared libraries as aggressively as static libraries. Shared libraries are compiled using position-independent code, which limits some opportunities for optimization, while our static libraries are not.

To link with static libraries instead of shared libraries use the `-static` option. For example the following code is linked using the shared libraries.

```
$ pathcc -o hello hello.c
$ ldd hello
       libpscrt.so.1 => /opt/pathscale/lib/1.2/libpscrt.so.1
(0x0000002a9566d000)
       libmpath.so.1 => /opt/pathscale/lib/1.2/libmpath.so.1
(0x0000002a9576e000)
       libc.so.6 => /lib64/libc.so.6
(0x0000002a9588b000)
       libm.so.6 => /lib64/libm.so.6
(
0x0000002a95acd000)
       /lib64/ld-linux-x86-64.so.2 => /lib64/ld-linux-x86-64.so.2
(0x0000002a95556000) $
```

If you use the `-static` option, notice that the shared libraries are no longer required.

```
$ pathcc -o hello hello.c -static
$ ldd hello
       not a dynamic executable
$
```

2.7 Large file support

The Fortran runtime libraries are compiled with large file support. PathScale does not provide any runtime libraries for C or C++ that do I/O, so large file support is provided by the libraries in the Linux distribution being used.

2.8 Large object support

The PathScale compilers currently support two memory models: small and medium.

The default memory model on AMD64 systems, and the default for the compilers, is small (equivalent to GCC's `-mcmodel=small`). This means that offsets of code and data within binaries are represented as signed 32-bit quantities. In this model, all code and data in an executable must come to less than 2GB in total size. Note that by data, we mean static and unlimited static data that are compiled into an executable, not data allocated dynamically on the stack or from the heap.

Pointers are 64-bits however, so dynamically allocated memory may exceed 2GB. Programs can be statically or dynamically linked.

Additionally the compilers support the medium memory model with the use of the option `-mcmodel=medium` on all of the compilation and link commands. This means that offsets of code within binaries are represented as signed 32-bit quantities. The offsets for data within the binaries are represented as signed 64-bit quantities. In this model, all code in an executable must come to less than 2GB in total size. The data, both static and BSS, are allowed to exceed 2GB in size.

As with the small memory model, pointers are also signed 64-bit quantities and may exceed 2 GB in size.

See 9.3 for more information on using large objects, and your GCC 3.3.1 documentation for more information on this topic.

2.8.1 Support for "large" memory model

At this time the PathScale compilers do not support the large memory model. The significance is that the code offsets must fit within the signed 32-bit address space. To determine if you are close to this limit, use the Linux `size` command.

```
$ size bench
   text    data     bss     dec     hex filename
 910219    1448    3192   914859   df5ab bench
```

If the total value of the `text` segment is close to 2GB, then this may be an issue for you. We believe that codes that are this large are extremely rare and would like to know if you are using such an application.

The size of the `bss` and `data` segments are addressed by using the medium memory model.

2.9 Debugging

The flag '`-g`' tells the PathScale EKO compilers to produce data in the form used by modern debuggers, such as GDB. This format is known as DWARF 2.0 and is incorporated directly into the object files. Code that has been compiled using `-g` will be capable of being debugged using GDB or other debuggers.

The `-g` option automatically sets the optimization level to `-O0` unless an explicit optimization level is provided on the command line. Debugging of higher levels of optimization is possible, but the code transforming performed by the optimizations may make it more difficult.

See the individual chapters on the PathScale EKO Fortran and C /C++ compilers for more language-specific debugging information, and Section 9 for debugging and troubleshooting tips.

2.10 Profiling: Locate your program's hot spots

To figure out where to tune your code, use `time` for a rough estimate to see if the issue is system load, application load, or a system resource, and `pathprof` to find the program's hot spots.

NOTE: The `pathprof` program is the complimentary version of `gprof` included in the PathScale EKO Compiler Suite.

The `time` tool provides the elapsed (or `wall`) time, user time, and system time of your program. Its usage is typically: `time ./program args`. Elapsed time is the measure of interest, especially for parallel programs, but if your system is busy with other loads, then user time would usually be a more accurate estimate of performance than elapsed time. If there is substantial system time and you don't expect to be using substantial non-compute resources of the system, you should use a kernel profiling tool to see what is causing it.

Often a program has "hot spots," a few routines or loops that are responsible for most of the execution time. Profilers are a common tool for finding the hot spots of a program. Once you find the hot spots in your program, you can concentrate on either improving your code for better performance, or you may get some insight into which compiler flags are likely to lead to better performance.

The PathScale EKO Compiler Suite includes a version of the standard Linux profiler `gprof` (`pathprof`). There are more details and an example later in Chapter 8, but the following steps are all that are needed to get started in profiling:

1. Add the `-pg` flag to both the compile and link steps with the PathScale EKO compilers. This generates an instrumented binary.

2. Run the program executable with the input data of interest. This creates a `gmon.out` file with the profile data.

3. Run `pathprof <program-name>` to generate the profiles. The standard output of `pathprof` includes two tables:

   (a) a flat profile with the time consumed in each routine and the number of times it was called, and

   (b) a call-graph profile that shows, for each routine, which routines it called and which other routines called it. There is also an estimate of the inclusive time spent in a routine and all of the routines called by that routine.

See Section 8 for a more detailed example of profiling.

2.11 Taskset: Assigning a process to a specific CPU

To improve the performance of the compiler on multiprocessor machines, it is often useful to assign the process to a specific CPU. The tool used to do this is `taskset`, which can be used to retrieve or set a process' affinity. This command is part of the `schedutils` package/RPM and may or may not be installed as part of your default configuration.

The CPU affinity is represented as a bitmask, typically given in hexadecimal. Assigning a process to a specific CPU prevents the Linux scheduler from moving or splitting the process.

Example:

```
$ taskset 0x00000001
```

This would assign the process to processor #0.

If an invalid mask is given, an error is returned, so when `taskset` returns, it is guaranteed that the program has been scheduled on a valid and legal CPU. See the `taskset(1)` man page for more information.

NOTE: Some of the Linux distributions supported by the PathScale compilers do not contain the `schedutils` package/RPM.

Chapter 3

The PathScale EKO Fortran compiler

The PathScale EKO Fortran compiler supports Fortran 77, Fortran 90, and Fortran 95. The PathScale EKO Fortran compiler:

- Conforms to ISO/IEC 1539:1991 Programming languages–Fortran (Fortran 90)
- Conforms to the more recent ISO/IEC 1539-1:1997 Programming languages–Fortran (Fortran 95)
- Supports legacy FORTRAN 77 (ANSI X3.9-1978) programs
- Provides support for some common extensions to the above language definitions
- Links binaries generated with the GNU Fortran 77 compiler
- Generated code complies with AMD64 ABI

3.1 Using the Fortran compiler

To invoke the PathScale EKO Fortran compiler, use this command:

```
$ pathf90
```

By default, the compiler will treat input files with an `.F` suffix or `.f` suffix as fixed-form files. Files with an `.F90` suffix or `.f90` suffix are treated as free-form files. This behavior can overridden using the `-fixedform` and `-freeform` switches. See Section 3.1.1 for more information on fixed-form and free-form files.

Files ending in `.F90` or `.F` are first preprocessed using the Fortran preprocessor. If you specify the `-ftpp` option, all files are preprocessed using the Fortran preprocessor, regardless of suffix. See Section 3.4.1 for more information on preprocessing.

Invoking the compiler without any options instructs the compiler to use optimization level `-O2`. These three commands are equivalent:

```
$ pathf90 test.f90
$ pathf90 -O test.f90
$ pathf90 -O2 test.f90
```

Using optimization level `-O0` instructs the compiler to do no optimization. Optimization level `-O1` performs only local optimization. Level `-O2`, the default, performs extensive optimizations that will always shorten execution time, but may cause compile time to be lengthened. Level `-O3` performs aggressive optimization that may or may not improve execution time. See Section 7.1 for more information about the `-O` flag.

Use the `-ipa` switch to enable inter-procedural analysis:

```
$ pathf90 -c -ipa matrix.f90
$ pathf90 -c -ipa prog.f90
$ pathf90 -ipa matrix.o prog.o -o prog
```

Note that the link line also specifies the `-ipa` option. This is required to perform the IPA link properly.

See Section 7.3 for more information on IPA.

NOTE: The compiler typically allocates data for Fortran programs on the stack for best performance. Some major Linux distributions impose a relatively low limit on the amount of stack space a program can use. When you attempt to run a Fortran program that uses a large amount of data on such a system, it will print an informative error message and abort. You can use your shell's "`ulimit`" (bash) or "`limit`" (tcsh) command to increase the stack size limit to a point where the program no longer crashes, or remove the limit entirely. See Section 5.5 for more information on this subject.

3.1.1 Fixed-form and free-form files

Fixed-form files follow the obsolete Fortran standard of assigning special meaning to the first 6 character positions of each line in a source file.

If a `C`, `!` or `*` character is present in the first character position on a line, that specifies that the remainder of the line is to be treated as a comment. If a `!` is present at any

character position on a line except for the 6th character position, then the remainder of that line is treated as a comment. Lines containing only blank characters or empty lines are also treated as comments.

If any character other than a blank character is present in the 6th character position on a line, that specifies that the line is a continuation from the previous line. The Fortran standard specifies that no more than 19 continuation lines can follow a line, but the PathScale compiler supports up to 499 continuation lines.

Source code appears between the 7th character position and the 72nd character position in the line, inclusive. Semicolons are used to separate multiple statements on a line. A semicolon cannot be the first non-blank character between the 7th character position and the 72nd character position.

Character positions 1 through 5 are for statement labels. Since statement labels cannot appear on continuation lines, the first five entries of a continuation line must be blank.

Free-form files have fewer limitations on line layout. Lines can be arbitrarily long, and continuation is indicated by placing an ampersand (&) at the end of the line before the continuation line. Statement labels can be placed at any character position in a line, as long as it is preceded by blank characters only. Comments start with a ! character anywhere on the line.

3.2 Modules

When a Fortran module is compiled, information about the module is placed into a file called `MODULENAME.mod` in the directory where the command is executed. This file allows other Fortran files to use procedures, functions, variables, and any other entities defined in the module. Module files can be considered similar to C header files. Like C header files, you can use the `-I` option to point to the location of module files:

```
$ pathf90 -I/work/project/include -c foo.f90
```

This instructs the compiler to look for `.mod` files in the `/work/project/include` directory. If `foo.f90` contains a 'use arith' statement, the following locations would be searched:

```
/work/project/include/ARITH.mod
./ARITH.mod
```

3.3 Extensions

The PathScale EKO Fortran compiler supports a number of extensions to the Fortran standard, which are described in this section.

3.3.1 Promotion of REAL and INTEGER types

Section 5 has more information about porting code, but it useful to mention the following option you can use to help in porting your Fortran code.

`-r8 -i8` Respectively promotes the default representation for REAL and INTEGER type from 4 bytes to 8 bytes. Useful for porting from Cray code when integer and floating point data is 8 bytes long by default. Watch out for type mismatches with external libraries.

NOTE: The `-r8` and `-i8` flags only affect default reals and integers, not variable declarations or constants which specify an explicit KIND. This can cause incorrect results if a 4-byte default real or integer is passed into a subprogram which declares a `KIND=4` integer or real. Using an explicit KIND value like this is unportable and is not recommended. Correct usage of KIND (i.e. `KIND=4`) will not result in any problems.

3.3.2 Cray pointers

The Cray pointer is a data type extension to Fortran to specify dynamic objects, different from the Fortran pointer. Both Cray and Fortran pointers use the `POINTER` keyword, but they are specified in such a way that the compiler can differentiate between them.

The declaration of a Cray pointer is:

```
POINTER ( <pointer>, <pointee> )
```

Fortran pointers are declared using:

```
POINTER :: [ <object_name> ]
```

PathScale's implementation of Cray Pointers is the Cray implementation, which is a stricter implementation than in other compilers. In particular, the PathScale EKO Fortran compiler does not treat pointers exactly like integers. The compiler will report an error if you do something like `p = ((p+7)/8)*8` to align a pointer.

3.3.3 Directives

At this time the PathScale compiler does not support directives. We will be evolving support for them in future releases.

3.4 Compiler and runtime features

3.4.1 Preprocessing source files

Before being passed to the compiler front-end, source files are optionally passed through a source code preprocessor. The preprocessor searches for certain directives in the file and, based on these directives, can include or exclude parts of the source code, include other files or define and expand macros.

All Fortran `.F` and `.F90` files are passed through the Fortran preprocessor, which is the same as the C processor with the `-traditional` flag used. No `.f` or `.f90` files are passed through the preprocessor unless the `-ftpp` flag is used.

3.4.2 Explain

The `explain` program is a compiler and runtime error message utility that prints a more detailed message for the numerical compiler messages you may see.

When the Fortran compiler or runtime prints out an error message, it prefixes the message with a string in the format “subsystem-number”. For example, “`pathf90-0724`”. The “`pathf90-0724`” is the message ID string that you will give to `explain`.

When you type `explain pathf90-0724`, the `explain` program provides a more detailed error message:

```
$ explain pathf90-0724
Error : Unknown statement. Expected assignment statement
but found "%s" instead of "=" or "=>".

The compiler expected an assignment statement
but could not find an assignment or pointer assignment
operator at the correct point.
```

Another example:

```
$ explain pathf90-0700
Error : The intrinsic call "%s" is being made with illegal
arguments.

A function or subroutine call which invokes the name of
an intrinsic procedure does not match any specific intrinsic.
All dummy arguments without the OPTIONAL attribute must
match in type and rank exactly.
```

3.4.3 Mixed code

If you have a large application that mixes Fortran code with code written in other languages, and the main entry point to your application is from C or C++, you can optionally use `pathcc` or `pathCC` to link the application, instead of `pathf90`. If you do, you must manually add the Fortran runtime libraries to the link line.

As an example, you might do something like this:

```
$ pathCC -o my_big_app file1.o file2.o -lpathfortran
```

3.4.4 Bounds checking

The PathScale EKO Fortran compiler can perform bounds checking on arrays. To enable this feature, use the `-C` option:

```
$ pathf90 -C gasdyn.f90 -o gasdyn
```

The generated code checks all array accesses to ensure that they fall within the bounds of the array. If an access falls outside the bounds of the array, you will get a warning from the program printed on the standard error at runtime:

```
$ ./gasdyn
lib-4961 : WARNING
  Subscript 20 is out of range for dimension 1 for array
  'X' at line 11 in file 't.f90' with bounds 1:10.
```

If you set the environment variable `F90_BOUNDS_CHECK_ABORT` to `YES`, then the resulting program will abort on the first bounds check violation.

Obviously, array bounds checking will have an impact on code performance, so it should be enabled only for debugging and disabled in production code that is performance sensitive.

3.4.5 Pseudo-random numbers

The pseudo-random number generator (PRNG) implemented in the standard PathScale EKO Fortran library is a non-linear additive feedback PRNG with a 32-entry long seed table. The period of the PRNG is approximately 16*((2**32)-1).

3.5 Runtime I/O compatibility

Files generated by the Fortran I/O libraries on other systems may contain data in different formats than that generated or expected by codes compiled by the PathScale EKO Fortran compiler. This section discusses how the PathScale EKO Fortran compiler interacts with files created by other systems.

3.5.1 Performing endian conversions

Use the assign command, or the `ASSIGN()` procedure, to perform endian conversions while doing file I/O.

3.5.1.1 The assign command

The `assign` command changes or displays the I/O processing directives for a Fortran file or unit. The `assign` command allows various processing directives to be associated with a unit or file name. This can be used to perform numeric conversion while doing file I/O.

The `assign` command uses the file pointed to by the `FILENV` environment variable to store the processing directives. This file is also used by the Fortran I/O libraries to load directives at runtime. See the `assign(1)` man page for more details and information.

For example:

```
$ FILENV=.assign
$ export FILENV
$ assign -N mips u:15
```

This instructs the Fortran I/O library to treat all numeric data read from or written to unit 15 as being MIPS-formatted data. This effectively means that the contents of the file will be translated from big-endian format (MIPS) to little-endian format (Intel) while being read. Data written to the file will be translated from little-endian format to big-endian format.

3.5.1.2 Using the wildcard option

The wildcard option for the `assign` command is:

```
assign -N mips p:%
```

Before running your program, run the following commands:

```
$ FILENV=.assign
$ export FILENV
$ assign -N mips p:%
```

3.5.1.3 Converting data and record headers

To convert numeric data in all unformatted units from big endian, and convert the record headers from big endian, use the following:

```
$ assign -F f77.mips -N mips g:su
$ assign -I -F f77.mips -N mips g:du
```

3.5.1.4 The ASSIGN() procedure

The `ASSIGN()` procedure provides a programmatic interface to the assign command. It takes as an argument a string specifying the assign command and an integer to store a returned error code. For example:

```
integer :: err
call ASSIGN("assign -N mips u:15", err)
```

This example has the same effect as the example in Section 3.5.1.1.

3.6 Source code compatibility

This section discusses our compatibility with source code developed for other compilers. Different compilers represent types in various ways, and this may cause some problems.

3.6.1 Fortran KINDs

The Fortran KIND attribute is a way to specify the precision or size of a type. Modern Fortran uses "kinds" to declare types. This system is very flexible, but has one drawback. The recommended and portable way to use KINDS is to find out what they are like this:

```
integer :: dp_kind = kind(0.0d0)
```

In actuality, some users hard-wire the actual values into their programs:

```
integer :: dp_kind = 8
```

This is an unportable practice, because some compilers use different values for the KIND of a double-precision floating point value.

The majority of compilers use the number of bytes in the type as the KIND value. For floating point numbers, this means KIND=4 is 32-bit floating point, and KIND=8 is 64-bit floating point. The PathScale compiler follows this convention.

Unfortunately for us and our users, this is incompatible with unportable programs written using GNU Fortran, `g77`. `g77` uses KIND=1 for single precision (32 bits) and KIND=2 for double precision (64 bits). For integers, however, `g77` uses KIND=3 for 1 byte, KIND=5 for 2 bytes, KIND=1 for 4 bytes, and KIND=2 for 8 bytes.

We are investigating the cost of providing a compatibility flag for unportable `g77` programs. If you find this to be a problem, the best solution is to change your program to inquire for the actual KIND values instead of hard-wiring them.

3.6.2 Fortran 95

The PathScale EKO Fortran compiler is compliant with the Fortran 95 standard. The only outstanding issue as of release 1.2 is that initializing `POINTER` elements of derived types to `NULL()` incorrectly gives an error. This feature is expected to be implemented soon.

3.7 Library compatibility

This section discusses our compatibility with libraries compiled with C or other Fortran compilers.

Linking object code compiled with other Fortran compilers is a complex issue. Fortran 90 or 95 compilers implement modules and arrays so differently that it is hopeless to attempt to link code from two or more compilers. For Fortran 77, run-time libraries for things like I/O and intrinsics are different, but it is possible to link both runtime libraries to an executable. We have experimented with this with object code compiled by `g77`, and it works at least some of the time. It is possible that some of our library functions have the same name but different calling conventions than some of `g77`'s library functions. We have not experimented at all with linking to object code from the PGI or Intel compilers.

3.7.1 Name mangling

Name mangling is a mechanism by which names of functions, procedures, and common blocks from Fortran source files are converted into an internal representation when compiled into object files. For example, a Fortran subroutine called `foo` gets turned into the name "`foo_`" when placed in the object file. We do this to avoid name collisions with similar functions in other libraries. This makes mixing code from C, C++, and Fortran easier.

Name mangling ensures that function, subroutine, and common-block names from a Fortran program or library do not clash with names in libraries from other programming languages. For example, the Fortran library contains a function named "`access`", which performs the same function as the function `access` in the standard C library. However, the Fortran library `access` function takes four arguments, making it incompatible with the standard C library `access` function, which takes only two arguments. If your program links with the standard C library, this would cause a symbol name clash. Mangling the Fortran symbols prevents this from happening.

By default, we follow the same name mangling conventions as the GNU `g77` compiler and `libf2c` library when generating mangled names. Names without an underscore have a single underscore appended to them, and names containing an underscore have two underscores appended to them. The following examples should help make this clear:

```
molecule -> molecule_
run_check -> run_check__
energy_ -> energy___
```

This behavior can be modified by using the `-fno-second-underscore` and the `-fno-underscoring` options to the `pathf90` compiler.

PGI Fortran and Intel Fortran's default policies correspond to our `-fno-second-underscore` option.

Common block names are also mangled. Our name for the blank common block is the same as `g77` (`_BLNK__`). PGI's compiler uses the same name for the blank common block, while Intel's compiler uses `_BLANK__`.

3.7.2 ABI compatibility

The PathScale EKO compilers support the official AMD64 Application Binary Interface (ABI), which is not always followed by other compilers. In particular, `g77` does not pass the return values from functions returning `COMPLEX` or `REAL` values according to the AMD64 ABI. (Double precision `REAL`s are OK.) For more details about what `g77` does, see the "`info g77`" entry for the `-ff2c` flag.

This issue is a problem when linking binary-only libraries such as Kazushige Goto's BLAS library or the ACML library (AMD Core Math Library). Libraries such as FFTW and MPICH don't have any functions returning REAL or COMPLEX, so there are no issues with these libraries.

For linking with g77-compiled functions returning COMPLEX or REAL values see Section 3.7.3.

Like most Fortran compilers, we represent character strings passed to subprograms with a character pointer, and add an integer length parameter to the end of the call list.

3.7.3 Linking with g77-compiled libraries

If you wish to link with a library compiled by g77, and if that library contains functions that return COMPLEX or REAL types, you need to tell the PathScale compiler to treat those functions differently. Use the -ff2c-abi switch to point the PathScale compiler at a file that contains a list of functions in the g77-compiled libraries that return COMPLEX or REAL types. When the PathScale compiler generates code that calls these listed functions, it will modify its ABI behavior to match g77's expectations.

NOTE: You can only specify the -ff2c-abi switch once on the command line. If you have multiple g77-compiled libraries, you need to place all the appropriate symbol names into a single file.

The format of the file is one symbol per line. Each symbol should be as you would specify it in your Fortran code (i.e. do not mangle the symbol). As an example:

```
$ cat example-list
sdot
cdot
$
```

You can use the fsymlist program to generate a file in the appropriate format. For example:

```
$ fsymlist /opt/acml2.0/gnu64/lib/libacml.a > acml-2.0-list
```

This will find all Fortran symbols in the libacml.a library and place them into the acml-2.0-list file. You can then use this file with the -ff2c-abi switch. See Section 3.7.3.1 for more details on using the switch with ACML.

NOTE: The fsymlist program generates a list of *all* Fortran symbols in the library, including those that do not return COMPLEX or REAL types. The extra symbols will be ignored by the compiler.

3.7.3.1 AMD Core Math Library (ACML)

The AMD Core Math Library (ACML) incorporates BLAS, LAPACK, and FFT routines, and is designed to obtain excellent performance from applications running on AMD platforms. This highly optimized library contains numeric functions for mathematical, engineering, scientific, and financial applications. ACML is available both as a 32-bit library (for compatibility with legacy x86 applications), and as a 64-bit library that is designed to fully exploit the large memory space and improved performance offered by the AMD64 architecture.

There are two issues to be solved: An I/O library issue, and an ABI issue. In order to link to the gcc/g77 version of the ACML library, you need to link to g77's I/O library. You can do this by adding `-lg2c` to your link line.

For the ABI issue, you need the `-ff2c-abi` switch in all your compilations. We have provided a symbol list to use for both the ACML 1.5 and ACML 2.0 libraries (`acml-1.5` and `acml-2.0`).

To use ACML 1.5 with the PathScale EKO Fortran compiler, use the following:

```
$ pathf90 -ff2c-abi /opt/pathscale/etc/f2c-abi/acml-1.5
    foo.f bar.f
```

You should then link with the GNU version of the ACML libraries:

```
$ pathf90 -o program foo.o bar.o -lacml -lg2c
```

To use ACML 2.0 with the PathScale EKO Fortran compiler, use the following:

```
$ pathf90 -ff2c-abi /opt/pathscale/etc/f2c-abi/acml-2.0
    foo.f bar.f
```

3.8 Debugging and troubleshooting

The flag '`-g`' tells the PathScale EKO compilers to produce data in the form used by modern debuggers, such as GDB, Etnus' TotalView, Absoft Fx2, and Streamline's DDT. This format is known as DWARF 2.0 and is incorporated directly into the object files. Code that has been compiled using `-g` will be capable of being debugged using GDB or other debuggers.

The `-g` option automatically sets the optimization level to `-O0` unless an explicit optimization level is provided on the command line. Debugging of higher levels of

optimization is possible, but the code transforming performed by the optimizations many make it more difficult.

Bounds checking is quite a useful debugging aid. This can also be used to debug allocated memory.

If you are noticing numerical accuracy problems, see Section 7.7 for more information on numerical accuracy.

See Section 9 for more information on debugging and troubleshooting.

3.8.1 Writing to constants can cause crashes

Some Fortran compilers allocate storage for constant values in read-write memory. The PathScale EKO Fortran compiler allocates storage for constant values in read-only memory. Both strategies are valid, but the PathScale compiler's approach allows it to propagate constant values aggressively.

This difference in constant handling can result in crashes at runtime when Fortran programs that write to constant variables are compiled with the PathScale EKO Fortran compiler. A typical situation is that an argument to a subroutine or function is given a constant value such as `0` or `.FALSE.`, but the subroutine or function tries to assign a new value to that argument.

We recommend that where possible, you fix code that assigns to constants so that it no longer does this. Such a change will continue to work with other Fortran compilers, but will allow the PathScale EKO Fortran compiler to generate code that will not crash and will run more efficiently.

If you cannot modify your code, we provide an option called `-LANG:rw_const=on` that will change the compiler's behavior so that it allocates constant values in read-write memory. We do not make this option the default, as it reduces the compiler's ability to propagate constant values, which makes the resulting executables slower.

3.8.2 Aliasing: -OPT:alias=no_parm

The Fortran standards require that arguments to functions and subroutines not alias each other. As an example, this is illegal:

```
program bar
...
call foo(c,c)
...
subroutine foo(a,b)
integer i
```

```
real a(100), b(100)
do i = 2, 100
  a(i) = b(i) - b(i-1)
enddo
```

In this example, if the dummy arguments `a` and `b` are actually the same array, `foo` will get the wrong answer due to aliasing.

Programmers occasionally break this aliasing rule, and as a result, their programs get the wrong answer under high levels of optimization. This sort of bug frequently is thought to be a compiler bug, so we have added this option to the compiler for testing purposes. If your program gets the right answer with `-OPT:alias=no_parm`, and the wrong answer without, then your program is breaking the aliasing rule.

Chapter 4

The PathScale EKO C/C++ compiler

The PathScale EKO C and C++ compilers conform to the following set of standards and extensions.

The C compiler:

- Conforms to ISO/IEC 9899:1990, Programming Languages - C standard
- Supports extensions to the C programming language as documented in "*Using GCC: The GNU Compiler Collection Reference Manual*," October 2003, for GCC version 3.3.1.
- Refer to section 4.4.1 of this document for the list of extensions that are currently not supported.
- Complies with the C Application Binary Interface as defined by the GNU C compiler (`gcc`) as implemented on the platforms supported by the PathScale EKO Compiler Suite.
- Supports most of the widely used command line options supported by `gcc`.
- Generated code complies with AMD64 ABI

The C++ compiler:

- Conforms to ISO/IEC 14882:1998(E), Programming Languages - C++ standard
- Supports extensions to the C++ programming language as documented in "*Using GCC: The GNU Compiler Collection Reference Manual*," October 2003, for GCC version 3.3.1. Refer to section 4.4.1 of this document for the list of extensions that are currently not supported.

- Complies with the C Application Binary Interface as defined by the GNU C++ compiler (`g++`) as implemented on the platforms supported by the PathScale EKO Compiler Suite.
- Supports most of the widely used command line options supported by `g++`.
- Generated code complies with AMD64 ABI

To invoke the PathScale EKO C and C++ compilers, use these commands:

- `pathcc` - invoke the C compiler
- `pathCC` - invoke the C++ compiler

The command line flags for both compilers are compatible with those taken by the GCC suite. See Section 4.1 for more discussion of this.

4.1 Using the C/C++ compilers

If you currently use the GCC compilers, the PathScale EKO compiler commands will be familiar. Makefiles that presently work with GCC should operate with the PathScale EKO compilers effortlessly–simply change the command used to invoke the compiler and rebuild. See Section 5.3.2 for information on modifying existing scripts

The invocation of the compiler is identical to the GCC compilers, but the flags to control the compilation are different. We have sought to provide flags compatible with GCC's flag usage whenever possible and also provide optimization features that are absent in GCC, such as IPA and LNO.

Generally speaking, instead of being a single component as in GCC, the PathScale compiler is structured into components that perform different classes of optimizations. Accordingly, compilation flags are provided under group names like `-IPA`, `-LNO`, `-OPT`, `-CG`, etc. For this reason, many of the compilation flags in PathScale will differ from those in GCC. See the list of optimization flags in Appendix **??** for more information.

The default optimization level is 2. This is equivalent to passing `-O2` as a flag. The following three commands are identical in their function:

```
$ pathcc hello.c
$ pathcc -O hello.c
$ pathcc -O2 hello.c
```

See Section 7.1 for information about the optimization levels available for use with the compiler.

To run with `-Ofast` or with `-ipa`, the flag must also be given on the link command.

```
$ pathCC -c -Ofast warpengine.cc
$ pathCC -c -Ofast wormhole.cc
$ pathCC -o ftl -Ofast warpengine.o wormhole.o
```

See Section 7.3 for information on `-ipa` and `-Ofast`.

4.2 Compiler and runtime features

4.2.1 Preprocessing source files

Before being passed to the compiler front-end, source files are optionally passed through a source code preprocessor. The preprocessor searches for certain directives in the file and, based on these directives, can include or exclude parts of the source code, include other files or define and expand macros.

All C and C++ files are passed through the the C preprocessor unless the `-noccp` flag is specified.

4.2.2 Mixing code

If you have a large application that mixes Fortran code with code written in other languages, and the main entry point to your application is from C or C++, you can optionally use `pathcc` or `pathCC` to link the application, instead of `pathf90`. If you do, you must manually add the Fortran runtime libraries to the link line.

See Section 3.4.3 for details. To link object files that were generated with `pathCC` using `pathcc` or `pathf90`, include the option `-lstdc++`.

4.2.3 Linking

Note that the `pathcc` (C language) user needs to add `-lm` to the link line when calling `libm` functions. The second pass of feedback compilation may require an explicit `-lm`.

4.3 Debugging

The flag '`-g`' tells the PathScale EKO C and C++ compilers to produce data in the form used by modern debuggers, such as GDB. This format is known as DWARF 2.0 and is incorporated directly into the object files. Code that has been compiled using `-g` will be capable of being debugged using GDB or other debuggers.

The `-g` option automatically sets the optimization level to `-O0` unless an explicit optimization level is provided on the command line. Debugging of higher levels of optimization is possible, but the code transforming performed by the optimizations many make it more difficult.

See Section 9 for more information on troubleshooting and debugging.

4.4 GCC extensions not supported

The PathScale EKO C and C++ Compiler Suite supports most of the C and C++ extensions supported by GCC Version 3.3.1 Suite. In this release, we do not support the following extensions:

For C:

- Nested functions
- Complex integer data type: Complex integer data types are not supported. Although the PathScale EKO Compiler Suite fully supports floating point complex numbers, it does not support complex integer data types, such as `_Complex int`.
- Thread local storage
- Many of the `__builtin` functions
- Inline assembly
- A `goto` outside of the block. PathScale compilers do support taking the address of a label in the current function and doing indirect jumps to it.
- The compiler generates incorrect code for structs generated on the fly (a GCC extension).
- Currently we do not support pragmas; they will be supported in a future release.

For C++:

- Java-style exceptions
- `java_interface` attribute
- `init_priority` attribute
- Currently we do not support pragmas; they will be supported in a future release.

Chapter 5

Porting and compatibility

5.1 Getting started

Here are some tips to get you started compiling your favorite applications with the PathScale EKO Compiler Suite. Some of the known issues are:

- The PathScale EKO Compiler Suite C, C++, and Fortran compilers are compatible with `gcc` and `g77`. Some packages will check strings like the `gcc` version or the name of the compiler to make sure you are using `gcc`; you may have to work around these tests. See Section 5.3.1 for more information.

- Some packages continue to use deprecated features of `gcc`. While `gcc` may print a warning and continue compilation, the PathScale EKO Compiler Suite C, C++, and Fortran compilers may print an error and exit. Use the instructions in the error to substitute an updated flag. For example, some packages will specify the deprecated "`-Xlinker`" `gcc` flag to pass arguments to the linker, while the PathScale EKO Compiler Suite uses the modern "`-Wl`" flag.

- Some `gcc` flags may not yet be implemented. These will be documented in the release notes.

- If a `configure` script is being used, using the `compat-gcc` wrappers found in `<installation_dir>/compat-gcc/bin` may help. See Section 5.3.1 for more information.

- Some source packages make assumptions about the locations of libraries and fail to look in `lib64` named directories for libraries resulting in unresolved symbols at during the link.

5.2 Cookbook

This is a step-by-step approach to porting code. These are the steps to go through to get your code compiling with the PathScale EKO compilers:

1. Select sample code to work with
2. Change your makefile if necessary (very likely if you use IPA)
3. Check for these things:
   (a) Look for library dependencies
   (b) Check the options you are using
   (See the `eko` man page for the Pathscale compiler options)
   (c) Check extensions[1]
   (d) Check intrinsic functions
   (See Appendix B for the list of supported instrinsics)
4. Compile your sample code and look at the results
   (a) Look for behavior differences; does the program behave correctly?
   (b) Are you getting the right answer?
   (for example, with numerical analysis)
5. Troubleshoot and repeat

5.3 Compatibility

5.3.1 GCC compatibility wrapper script

Many software build packages check for the existence of `gcc`, and may even require the compiler used to be called `gcc` in order to build correctly. We provide a GCC compatibility wrapper script in `/opt/pathscale/compat-gcc/bin` (or `<install_directory>/compat-gcc/bin`).

This script can be invoked with different names:

- `gcc`, `cc` - to look like the GNU C compiler, and call `pathcc`
- `g++`, `c++` - to look like the GNU C++ compiler, and call `pathCC`
- `g77`, `f77` - to look like the GNU Fortran compiler, and call `pathf90`

[1] While the PathScale compilers aim to be compatible with code from other vendors, you may encounter unsupported extensions. All of the planned extensions for the compilers have not been implemented in this 1.2 release.

To use this script, you must put the path to this directory in your shell's search path *before* the location of your system's `gcc` (which is usually `/usr/bin`). You can confirm the order in the search path by running "`type gcc`" after modifying your search path. The output should print the location of the gcc wrapper, not `/usr/bin/gcc`.

To pass in PathScale-specific compiler options, you can set several environment variables before you do a build. They are:

`PSC_GENFLAGS` - generic flags, passed to all compilers

`PSC_CFLAGS` - only passed to the C compiler, `pathcc`

`PSC_CXXFLAGS` - only passed to the C++ compiler, `pathCC`

`PSC_FFLAGS` - only passed to the Fortran compiler, `pathf90`

5.3.2 Modifying existing scripts

If you are building a piece of software that is configured with GNU `autoconf`, you can run the configure script like this (using Bourne shell syntax):

```
$ CC=pathcc CXX=pathCC FC=pathf90 ./configure [..usual options..]
```

If you are using a regular Makefile, you may simply be able to run it as follows:

```
$ make CC=pathcc CXX=pathCC FC=pathf90
```

Software packages that build or configure in somewhat different ways, such as many scientific libraries, may need a little more work.

5.3.3 Name mangling

Name mangling ensures that function, subroutine, and common-block names from a Fortran program or library do not clash with names in libraries from other programming languages. This makes mixing code from C, C++, and Fortran easier. See Section 3.7.1 for details on name mangling.

5.4 Compiler options for porting and correctness

The following options can help you fix problems prior to debugging your code.

`-static` Some codes expect data to be initialized to zero and allocated in the heap.

`-r8 -i8` Respectively promotes the default representation for REAL and INTEGER type from 4 bytes to 8 bytes. Useful for porting from Cray code when integer and floating point data is 8 bytes long by default. Watch out for type mismatches with external libraries.

5.5 Fortran compiler stack size

The Fortran compiler allocates data on the stack by default. Some environments set a low limit on the size of a process's stack, which may cause Fortran programs that use a large amount of data to crash shortly after they start.

If the PathScale EKO Fortran runtime environment detects a low stack size limit, it will automatically increase the size of the stack allocated to a Fortran process before the Fortran program begins executing.

By default, it automatically increases this limit to the total amount of physical memory on a system, less 128 megabytes per CPU. For example, when run on a 4-CPU system with 1G of memory, the Fortran runtime will attempt to raise the stack size limit to 1G - (128M * 4), or 640M.

To have the Fortran runtime tell you what it is doing with the stack size limit, set the `PSC_STACK_VERBOSE` environment variable before you run a Fortran program. You can control the stack size limit that the Fortran runtime attempts to use using the `PSC_STACK_LIMIT` environment variable.

If this is set to the empty string, the Fortran runtime will not attempt modify the stack size limit in any way.

Otherwise, this variable must contain a number. If the number is not followed by any text, it is treated as a number of bytes. If it is followed by the letter "`k`" or "`K`", it is treated as kilobytes (1024 bytes). If "`m`" or "`M`", it is treated as megabytes (1024K). If "`g`" or "`G`", it is treated as gigabytes (1024M). If "`%`", it is treated as a percentage of the system's physical memory.

If the number is negative, it is treated as the amount of memory to leave free, i.e. it is subtracted from the amount of physical memory on the machine. If the text so far is followed by "`/cpu`", it is treated by a "per cpu" number, and the number is multiplied by the number of CPUs on the system. This is useful for multiprocessor systems that are running several processes concurrently.

For a 4-CPU system with 1G of memory, here are examples of the meanings of some values for stack size that could be set:

```
100000       100000 bytes
820K         820K (839680 bytes)
-0.25g       all but 0.25G, or 0.75G total
128M/cpu     128M per CPU, or 512M total
-10M/cpu     all but 10M per CPU (all but 40M total), or 0.96G total
```

If the Fortran runtime encounters problems while attempting to modify the stack size limit, it will print some warning messages, but will not abort.

Chapter 6

Tuning Quick Reference

This chapter provides some ideas for tuning your code's performance with the PathScale EKO compiler.

The following sections describe a small set of tuning options that are relatively easy to try, and often give good results. These are tuning options that do not require Makefile changes, or risk the correctness of your code results. More detail on these flags can be found in the next chapter, in Appendix **??**, and in the man pages.

6.1 Basic optimization

Here are some things to try first when optimizing your code.

For basic optimization, use the `-O` flag, which is equivalent to `-O2`. This is the first flag to think about using when tuning your code. After trying `-O`, try:
`-O2`
then ,
`-O3`
and then,
`-O3 -OPT:Ofast.`

For more information on the `-O` flags and `-OPT:Ofast`, see Section 7.1.

6.2 IPA

Inter-Procedural Analysis (IPA), invoked most simply with `-ipa`, is a compilation technique that analyzes an entire program. This allows the compiler to do

optimizations without regard to which source file the code appears in. IPA can improve performance significantly.

IPA can be used in combination with the other optimization flags. `-O3 -ipa` or `-O2 -ipa` will typically provide increased performance over the `-O3` or `-O2` flags alone. `-ipa` needs to be used both in the compile and in the link steps of a build. See Section 7.3 for more details on how to use `-ipa`.

6.3 Feedback directed optimization

Feedback-directed optimization uses a special instrumented executable to collect profile information about the program that is then used in later compilations to tune the executable.

See Section 7.6 for more information.

6.4 Aggressive optimization

The PathScale EKO compilers provide an extensive set of additional options to cover special case optimizations. The ones documented in Chapter 7 contain options that may significantly improve the speed or performance of your code.

This section briefly introduces some of the first tuning flags to try beyond `-O2` or `-O3`. Some of these options require knowledge of what the algorithms are and what coding style of the program require, otherwise they may impact the program's correctness. Some of these options depend on certain coding practices to be effective.

One word of caution: The PathScale EKO Compiler Suite, like all modern compilers, has a range of optimizations. Some produce identical program output to the non-optimized, some can change the program's behavior slightly. The first class of optimizations is termed "safe" and the second "unsafe". See for Section 7.7 for more information on these optimizations.

`-OPT:Olimit=0` is a generally safe option but may result in the compilation taking a long time or consuming large quantities of memory. This option tells the compiler to optimize the files being compiled at the specified levels no matter how large they are.

The option `-fno-math-errno` bypasses the setting of ERRNO in math functions. This can result in a performance improvement if the program does not rely on IEEE exception handling to detect runtime floating point errors.

Likewise, `-OPT:roundoff=2` allows for fairly extensive code transformations that may result in floating point round-off or overflow differences in computations. Refer to Section 7.7.3.2 and 7.7.3 for more information.

The option `-OPT:div_split=on` allows the conversion of `x/y` into `x*(recip(y))` which may result in less accurate floating point computations. Refer to Sections 7.7.3.2 and 7.7.3 for more information.

The `-OPT:alias` settings allow the compiler to apply more aggressive optimizations to the program. The option `-OPT:alias=typed` assumes that the program has been coded in adherence with the ANSI/ISO C standard which states that two pointers of different types cannot point to the same location in memory. Setting `-OPT:alias=restrict` allows the compiler to assume that points refer to distinct, non-overlapping objects. If the these options are specified and the program does violate the assumptions being made, the program may behave incorrectly. Refer to Section 7.7.1 for more information.

There are several shorthand options that can be used in place of the above options. The option `-OPT:Ofast` is equivalent to `-OPT:roundoff=2:Olimit=0:div_split=on:alias=typed`. `-Ofast` is equivalent to `-O3 -ipa -OPT:fast -fno-math-errno`. When using this shorthand options, make sure the impact of the option is understood by stepwise building up the functionality by using the equivalent options.

There are many more options that may help the performance of the program. These options are discussed elsewhere in the User Guide and in the associated man pages.

6.5 Performance analysis

In addition to these suggestions for optimizing your code, here are some other ideas to assist you in tuning. Section 2.10 discusses figuring out where to tune your code, using `time` to get an overview of your code, and using `pathprof` to find your program's hot spots.

6.6 Optimize your hardware

Make sure you are optimizing your hardware as well. Section 7.8 discusses getting the best performance out of processors based on the AMD64 family of chips (Opteron, Athlon64, and Athlon64 FX).

Chapter 7

Tuning options

This chapter discusses in more depth some of the major groups of flags available in the PathScale EKO Compiler Suite.

7.1 Basic optimizations: The `-O` flag

The `-O` flag is the first flag to think about using. See Table 7.1 showing the default flag settings for various levels of optimization.

`-O0` (O followed by a zero) specifies no optimization–this is useful for debugging. The `-g` debugging flag is fully compatible with this level of optimization.

NOTE: Using `-g` by itself without specifying `-O` will change the default optimization level from `-O2` to `-O0` unless explicitly specified.

`-O1` specifies minimal optimizations with no noticeable impact on compilation time compared with `-O0`. Such optimizations are limited to those applied within straight-line code (basic blocks), like peephole optimizations and instruction scheduling. The `-O1` level of optimization minimizes compile time.

`-O2` only turns on optimizations which always increase performance and the increased compile time (compared to `-O1`) is commensurate with the increased performance. This is the default, if you don't use any of the `-O` flags. The optimizations performed at level `2` are:

- For inner loops, perform:
  - Loop unrolling
  - Simple if-conversion
  - Recurrence-related optimizations
- Two passes of instruction scheduling
- Global register allocation based on first scheduling pass
- Global optimizations within function scopes:
  - Partial redundancy elimination
  - Strength reduction and loop termination test replacement
  - Dead store elimination
  - Control flow optimizations
  - Instruction scheduling across basic blocks
- `-O2` implies the flag `-OPT:goto=on,` which enables the conversion of `GOTO`s into higher level structures like `FOR` loops.

`-O3` turns on additional optimizations which will most likely speed your program up, but may, in rare cases, slow your program down. The optimizations provided at this level include all `-O1` and `-O2` optimizations and the flags noted below:

- `-LNO:opt=1` Turn on Loop Nest Optimization (for more details, see Section 7.4)
- `-OPT` with the following options in the OPT group: (see the `-opt` man pages for more information)

  `-OPT:got_call_conversion=on` (see the `opt(7)` man page)

  `-OPT:roundoff=1` (see Section 7.7.3.2)

  `-OPT:IEEE_arith=2` (see Section 7.7.3)

  `-OPT:Olimit=6000` (see Section 6.3)

  `-OPT:reorg_common=1` (see the `opt(7)` man page)

NOTE: In our in-house testing, we have noticed that several codes which are slower at `-O3` than `-O2` are fixed by using `-O3 -LNO:prefetch=0`. This seems to mainly help codes that fit in cache.

7.2 Syntax for complex optimizations (-CG, -IPA, -LNO -OPT, -WOPT)

The group optimizations control a variety of behaviors and can override defaults. This section covers the syntax of these options.

The group options allow for the setting of multiple suboptions in two ways:

- Separating each sub-flag by colons, or
- Using multiple flags on the command line.

For example, the following command lines are equivalent:

```
pathcc -OPT:roundoff=2:alias=restrict wh.c
pathcc -OPT:roundoff=2 -OPT:alias=restrict wh.c
```

Some suboptions either enable or disable the feature. To enable a feature, either specify only the subflag name or with `=1`, `=ON`, or `=TRUE`. Disabling a feature, is accomplished by adding `=0`, `=OFF`, or `=FALSE`. The following command lines mean the same thing:

```
pathf90 -OPT:div_split:fast_complex=FALSE:IEEE_NaN_inf=OFF wh.F
pathf90 -OPT:div_split=1:fast_complex=0:IEEE_NaN_inf=false wh.F
```

7.3 Inter-Procedural Analysis (IPA)

IPA (Inter-Procedural Analysis) is a compilation technique that analyzes an entire program at once. It is most simply invoked with `-ipa`. IPA allows the compiler to do optimizations such as constant propagation and inlining of functions without regard to which source file code appears in. IPA can be used with any optimization level, but gives the biggest potential benefit when combined with `-O3`. The `-Ofast` flag turns on `-ipa` as part of its many optimizations.

Inter-procedural analysis is invoked in several possible ways: `-ipa`, `-IPA`, and implicitly via `-Ofast`. In the following section we briefly explain how to invoke this analysis, which can have a significant effect on performance.

When compiling with `-ipa` the `.o` files that are created are not a regular `.o` files. IPA uses the `.o` files in its analysis of your program, and then does a second compilation using that information to optimize the executable.

NOTE: When you are using `-ipa`, *all* the `.o` files have to have been compiled *with* `-ipa`, and *all* libraries have to have been compiled *without* `-ipa` for your compilation to be successful.

Currently the IPA linker is looking for one optimization level for the entire program. You will get a warning if there are several different levels of optimization in your compilation. The warning doesn't work with system libraries. In future versions of the compiler, you will be able to compile and link system libraries with different levels of optimization.

Flags like `-ipa` can be used in combination with a very large number of other flags, but some typical combinations with the `-O` flags are shown below:

`-O3 -ipa` or `-O2 -ipa` is a typical additional attempt at improved performance over the `-O3` or `-O2` flag alone. `-ipa` needs to be used both in the compile and in the link steps of a build.

Using IPA with your program can be simple or moderately complex. If you have only a few source files, you can simply use it like this:

```
pathf90 -O3 -ipa main.f subs1.f subs2.f
```

If you compile files separately, the `*.o` files generated by the compiler do not actually contain object code; they contain a representation of the source code. Actual compilation happens at link time. The link command also needs the `-ipa` flag added. Thus,

```
pathf90 -c -O3 -ipa main.f
pathf90 -c -O3 -ipa subs1.f
pathf90 -c -O3 -ipa subs2.f
pathf90 -O3 -ipa main.o subs1.o subs2.o
```

NOTE: IPA has some restrictions that may require modifying Makefiles. In particular, when you link, all `*.o` files must have been compiled with `-ipa`, and all library archives (`libfoo.a`) must have been compiled without `-ipa`. If your Makefiles build libraries, and you wish this code to be built with `-ipa`, you will need to split these libraries into separate `*.o` files before linking. For example, if your link line is:

```
pathf90 -O3 -ipa main.o sub1.o lib/libfoo.a
```

and the code in `libfoo.a` was built with IPA, you will need to do something like:

```
mkdir ipa_temp
cd ipa_temp
ar x ../lib/libfoo.a
cd ..
pathf90 -O3 -ipa main.o sub1.o ipa_temp/*.o
```

Note that, in a non-IPA compile, most of the time is incurred with compiling all the files to create the object files (the `.o`'s) and the link step is quite fast. In an IPA compile, the creating of `.o` files is very fast, but the link step can take a long time. The total compile time can be considerably longer with IPA than without.

7.3.1 Size and correctness limitations to IPA

IPA often works well on programs up to 100,000 lines, but is not recommended for use in larger programs in this release.

7.4 Loop Nest Optimization (LNO)

If your program has many nests of loops, you may want to try some of the Loop Nest Optimization group of flags. This group defines transformations and options that can be applied to loop nests.

One of the nice features of the PathScale EKO compilers is that its powerful Loop Nest Optimization feature is invoked by default at `-O3`. This feature can provide up to a 10-20x performance advantage over other compilers on certain matrix operations at `-O3`.

In rare circumstances, this feature can make things slower, so you can use `-LNO:opt=0` to disable nearly all loop nest optimization. Trying to make an `-O2` compile faster by adding `-LNO:opt=on` will not work because the `-LNO` feature is only active with `-O3` (or `-Ofast` which implies `-O3`).

Some of the features that one can control with the `-LNO:` group are:

- Loop fusion and fission
- Blocking to optimize cache line reuse
- Cache management
- TLB (Translation Lookaside Buffer) optimizations
- Prefetch

In this section we will highlight a few of the LNO options that have frequently been valuable.

7.4.1 Loop fusion and fission

Sometimes loop nests have too few instructions and consecutive loops should be combined to improve utilization of CPU resources. Another name for this process is loop fusion.

Sometimes a loop nest will have too many instructions, or deal with too many data items in its inner loop, leading to too much pressure on the registers, resulting in spills of registers to memory. In this case, splitting loops can be beneficial. Like splitting an atom, splitting loops is termed fission. These are the LNO options to control these transformations:

`-LNO:fusion=n` Perform loop fusion, `n:` `0` off, `1` conservative, `2` aggressive. Level 2 implies that outer loops in consecutive loop nests should be fused, even if it is found that not all levels of the loop nests can be fused. The default level is `1` (standard outer loop fusion), but `2` has been known to benefit a number of well-known codes.

`-LNO:fission=n` Perform loop fission, `n:` `0` off, `1` standard, `2` try fission before fusion. The default level is `1`, but `2` has been known to benefit a number of well-known codes.

Be careful with mixing the above two flags, because fusion has some precedence over fission: if `-LNO:fission=[1 or 2]` and `-LNO:fusion=[1 or 2]` then fusion is performed.

`-LNO:fusion_peeling_limit=n` controls the limit for the number of iterations allowed to be peeled in fusion, where `n` has a default of 5 but can be any non-negative integer.

Peeling is done when the iteration counts in consecutive loops is different, but close, and several iterations are replicated outside the loop body to make the loop counts the same.

7.4.2 Cache size specification

The PathScale EKO compilers are primarily targeted at the Opteron CPU currently, so they assume an L2 cache size of 1MB. Athlon 64 can have either a 512KB or 1MB L2 cache size. If your target machine is Athlon 64 and you have the smaller cache size, then setting `-LNO:cs2=512k` could help. Here is the more general description of some of what is available.

`-LNO:cs1=n`, `cs2=n`, `cs3=n`, `cs4=n`
This option specifies the cache size. `n` can be `0` or a positive integer followed by one of the following letters: `k`, `K`, `m`, or `M`. These letters specify the cache size in Kbytes or Mbytes.

Specifying `0` indicates there is no cache at that level.

`cs1` is the primary cache
`cs2` refers to the secondary cache
`cs3` refers to memory
`cs4` is the disk

Default cache size for each type of cache depends on your system. Use `-LIST:options=ON` to see the default cache sizes used during compilation.

With a smaller cache, the cache set associativity is often decreased as well. The flag set: `-LNO:assoc1=n`, `assoc2=n`, `assoc3=n`, `assoc4=n` can define this appropriately for your system.

Once again, the above flags are already set appropriately for Opteron.

7.4.3 Cache blocking, loop unrolling, interchange transformations

Cache blocking, also called 'tiling', is the process of choosing the appropriate loop interchanges and loop unrolling sizes at the correct levels of the loop nests so that cache reuse can be optimized and memory accesses reduced. This whole LNO feature is on by default, but can be turned off with: `-LNO:blocking=off`.
`-LNO:blocking_size=n` specifies a block size that the compiler must use when performing any blocking, where `n` is a positive integer that represents the number of iterations.

`-LNO:interchange` is on by default, but setting this `=0` can disable the loop interchange transformation in the loop nest optimizer.

The LNO group controls outer loop unrolling, but the `-OPT:` group controls inner loop unrolling. Here are the major `-LNO:` flags to control loop unrolling:

`-LNO:outer_unroll_max,ou_max=n` specifies that the compiler may unroll outer loops in a loop nest by up to `n` per loop, but no more. The default is `4`.

`-LNO:ou_prod_max=n`
Indicates that the product of unrolling levels of the outer loops in a given loop nest is not to exceed `n`, where `n` is a positive integer. The default is `16`.

To be more specific about how much unrolling is to be done, use `-LNO:outer_unroll,ou=n`. This indicates that exactly `n` outer loop iterations should be unrolled, if unrolling is legal. For loops where outer unrolling would cause problems, unrolling is not performed.

7.4.4 Prefetch

The LNO group can provide guidance to the compiler about the level and type of prefetching to enable. General guidance on how aggressively to prefetch is specified

by -LNO:prefetch=n, where n=1 is the default level. n=0 disables prefetching in loop nests, while n=2 means to prefetch more aggressively than the default.

-LNO:prefetch_ahead=n defines how many cache lines ahead of the current data being loaded should be prefetched. The default is n=2 cache lines.

7.4.5 Vectorization

Vectorization is an optimization technique that works on multiple pieces of data at once. For example, the compiler will turn a loop computing the mathematical function sin() into a call to the vsin() function, which is twice as fast.

The use of vectorized versions of functions in the math library like sin(), cosin() is controlled by the flag -LNO:vintr=ON|OFF. Vectorization of user code (excluding these mathematical functions) is controlled by the flag -LNO:simd[=(0|1|2)]. -LNO:simd_verbose=ON prints vectorizer information (from vectorizing user code) to stdout.

See the eko man page for more information.

7.5 Code Generation (-CG:)

The code generation group governs some aspects of instruction-level code generation that can have benefits for code tuning.

-CG:gcm=OFF turns off the instruction-level global code motion optimization phase. The default is ON.

-CG:load_exe=n specifies the threshold for subsuming a memory load operation into the operand of an arithmetic instruction. The value of 0 turns off this subsumption optimization. By default this subsumption is performed only when the result of the load has only one (n=1) use. This subsumption is not performed if the number of times the result of the load is used exceeds the value n, a non-negative integer. We have found that load_exe=2 or 0 are occasionally profitable.

-CG:use_prefetchnta=ON means for the compiler to use the prefetch operation that assumes that data is Non-Temporal at All (NTA) levels of the cache hierarchy. This is for data streaming situations in which the data will not need to be re-used soon. Default is OFF.

-CG:use_movlpd=ON makes the code generator use the MOVLPD SSE2 instruction instead of MOVSD. See AMD64's instruction description for the difference between these two instructions. Default is OFF.

7.6 Feedback Directed Optimization (FDO)

Feedback-directed optimization uses a special instrumented executable to collect profile information about the program; for example, it records how frequently every `if()` statement is true. This information is then used in later compilations to tune the executable.

FDO is most useful if a program's typical execution is roughly similar to the execution of the instrumented program on its input data set; if different input data has dramatically different `if()` frequencies, using FDO might actually slow down the program. This section also discusses how to invoke this feature with the `fb_create` and `fb_opt` flags.

FDO requires compiling the program at least twice. In the first pass:

```
pathcc -O3 -ipa -fb_create fbdata -o foo foo.c
```

The executable `foo` will contain extra instrumentation library calls to collect feedback information; this means `foo` will actually run a bit slower than normal.

Next, run the program `foo` with an example dataset:

```
./foo < typical_input_data >
```

During this run, a file named "`fbdata`" will be created, containing feedback information. To use this data in a subsequent compile:

```
pathcc -O3 -ipa -fb_opt fbdata -o foo foo.c
```

This new executable should run faster than a non-FDO `foo`, and will not contain any instrumentation library calls.

Experiment to see if FDO provides significant benefit for your application.

More details on feedback compilation with the PathScale EKO compilers can be found under the `-fb_create` and `-fb_opt` options in the `group_flags` man page.

7.7 Aggressive optimizations

The PathScale EKO Compiler Suite, like all modern compilers, has a range of optimizations. Some produce identical program output to the original, some can change the program's behavior slightly. The first class of optimizations is termed "safe" and the second "unsafe". As a general rule, our `-O1,-O2,-O3` flags only perform "safe" optimizations. But the use of "unsafe" optimizations often can produce a good speedup in a program, while producing a sufficiently accurate result.

Some "unsafe" optimizations may be "safe" depending on the coding practices used. We recommend first trying "safe" flags with your program, and then moving on to "unsafe" flags, checking for incorrect results and noting the benefit of unsafe optimizations.

Examples of unsafe optimizations include the following.

7.7.1 Alias analysis

Both C and Fortran have occasions where it's possible that two variables might occupy the same memory. For example, in C, two pointers might point to the same location, such that writing through one pointer changes the value of the variable pointed to by another. While the C standard prohibits some kinds of aliasing, many real programs violate these rules, so the aliasing behavior of PathScale's compiler is controlled by the `OPT:alias` flag. See Section 7.7.3.2 for more information.

Aliases are hidden definitions and uses of data due to:

- accesses through pointers
- partial overlap in storage locations (e.g. unions in C)
- procedure calls for non-local objects
- raising of exceptions

The compiler normally has to assume that aliasing will occur. The compiler does alias analysis to identify when there is no alias, so later optimizations can be performed. Certain C and C++ language rules allow some levels of alias analysis. Fortran has additional rules which make it possible to rule out aliasing in more situations: subroutine parameters have no alias; and side effects of calls are limited to global variables and actual parameters.

For C or C++, the coding style can help the compiler make the right assumptions. Using type qualifiers such as `const`, `restrict`, or `volatile` can help the compiler. Furthermore, if you supply some assumptions to make concerning your program, more optimizations can then be applied. The following are some of the various aliasing

models you can specify, listed in order of increasingly stringent, and potentially dangerous, assumptions you are telling the compiler to make about your program:

`-OPT:alias=any` the default level, which implies that any two memory references can be aliased.

`-OPT:alias=typed` means to activate the ANSI rule that objects are not aliased it they have different base types. This option is activated by `-Ofast`.

`-OPT:alias=unnamed` assumes that pointers never to point to named objects.

`-OPT:alias=restrict` tells the compiler to assume that all pointers are restricted pointers and point to distinct non-overlapping objects. This allows the compiler to invoke as many optimizations as if the program were written in Fortran. A restricted pointer behaves as though the C '`restrict`' keyword had been used with it in the source code.

`-OPT:alias=disjoint` says that any two pointer _expressions_ are assumed to point to distinct, non-overlapping objects.

To make the opposite assertion about your program's behavior, put '`no_`' before the value. For example, `-OPT:alias=no_restrict` means that distinct pointers may point to overlapping storage.

Additional `-OPT:alias` values are relevant to Fortran programmers in some situations:

`-OPT:alias=cray_pointer` asserts that an object pointed to by a Cray pointer is never overlaid on another variable's storage. This flag also specifies that the compiler can assume that the pointed-to object is stored in memory before a call to an external procedure and is read out of memory at its next reference. It is also stored before a `END` or `RETURN` statement of a subprogram.

`-OPT:alias=parm` promises that Fortran parameters do not alias to any other variable. This is the default. `no_parm` asserts that parameter aliasing is present in the program.

7.7.2 Numerically unsafe optimizations

Rearranging mathematical expressions and changing the order or number of floating point operations can slightly change the result. Example:

```
A = 2. * X
B = 4. * Y
C = 2. * (X + 2. * Y)
```

A clever compiler will notice that `C = A + B`. But the order of operations is different, and so a slightly different `C` will be the result. This particular transformation is controlled by the `OPT:roundoff` flag, but there are several other numerically unsafe flags.

Some options that fall into this category are:

The options that control IEEE behavior such as `-OPT:roundoff=N` and `-OPT:IEEE_arithmetic=N`. Here are a couple of others:

`-OPT:div_split=(ON|OFF)` This option enables or disables transforming expressions of the form `X/Y` into `X*(1/Y)`. The reciprocal is inherently less accurate than a straight division, but may be faster.

`-OPT:recip=(ON|OFF)` This option allows expressions of the form `1/X` to be converted to use the reciprocal instruction of the computer. This is inherently less accurate than a division, but will be faster.

These options can have performance impacts. For more information, see the `opt` manual page. You can view the manual page by typing `man opt` at the command line.

7.7.3 IEEE 754 compliance

It is possible to control the level of IEEE 754 compliance through options. Relaxing the level of compliance allows the compiler greater latitude to transform the code for improved performance. The following subsections discuss some of those options.

7.7.3.1 Arithmetic

Sometimes it is possible to allow the compiler to use operations that deviate from the IEEE 754 standard to obtain significantly improved performance, while still obtaining results that satisfy the accuracy requirements of your application.

The flag regulating the level of conformance to ANSI/IEEE 754-1985 floating pointing roundoff and overflow behavior is:

`-OPT:IEEE_arithmetic=N` (where `N= 1`, `2`, or `3`).

`-OPT:IEEE_arithmetic`

`=1` Requires strict conformance to the standard

`=2` Allows use of any operations as long as exact results are produced. This allows less accurate inexact results. For example, `X*0` may be replaced by `0`, and `X/X` may

replaced by `1` even though this is inaccurate when `X` is `+inf`, `-inf`, or `NaN`. This is the default level at `-O3`.

`=3` Means to allow any mathematically valid transformations. For example, replacing `x/y` by `x*(recip(y))`.

For more information on the defaults for IEEE arithmetic at different levels of optimization, see Table 7.1.

7.7.3.2 Roundoff

Use `-OPT:roundoff=` to identify the extent of roundoff error the compiler is allowed to introduce:

0 No roundoff error

1 Limited roundoff error allowed

2 Allow roundoff error caused by re-associating expressions

3 Any roundoff error allowed

The default roundoff level with `-O0`, `-O1`, and `-O2` is `0`. The default roundoff level with `-O3` is `2`.

Listing some of the other `-OPT:` suboptions that are activated by various roundoff levels can give more understanding about what the levels mean.

`-OPT:roundoff=1` implies:

- `-OPT:fast_exp=OFF` (it is `ON` at all other round off levels) This option enables optimization of exponentiation by replacing the run-time call for exponentiation by multiplication and/or square root operations for certain compile-time constant exponents (integers and halves).
- `-OPT:fast_trunc` implies inlining of the NINT, ANINT, AINT, and AMOD Fortran intrinsics.

`-OPT:roundoff=2` turns on the following sub-options:

- `-OPT:fold_reassociate` which allows optimizations involving re-association of floating-point quantities.
- `-OPT:recip` directs that faster, but potentially less accurate, reciprocal operations should be performed.

- `-OPT:rsqrt` tells the compiler to use faster, but potentially less accurate, square root operations.

`-OPT:roundoff=3` turns on the following sub-options:

- `-OPT:div_split` enables or disables the calculation of `x/y` as `x*(1.0/y)`.
- `-OPT:fast_complex` When this is set ON, complex absolute value (norm) and complex division use fast algorithms that overflow for an operand (the divisor, in the case of division) that has an absolute value that is larger than the square root of the largest representable floating-point number.
- `-OPT:fast_nint` uses a hardware feature to implement single and double-precision versions of NINT and ANINT

7.7.4 Other unsafe optimizations

A few advanced optimizations intended to exploit some exotic instructions such as CMOVE (conditional move) result in slightly changed program behavior, such as programs which write into variables guarded by an `if()` statement. For example:

```
if (a .eq. 1) then
   a = 3
endif
```

In this example, the fastest code on an x86 CPU is code which avoids a branch by always writing `a`; if the condition is false, it writes `a`'s existing value into `a`, else it writes `3` into `a`. If a is a read-only value not equal to `1`, this optimization will cause a segmentation fault in an odd but perfectly valid program.

7.7.5 Assumptions about numerical accuracy

See the following table for the assumptions made about numerical accuracy at different levels of optimization.

Table 7.1: **Numerical accuracy with options**

-OPT: option name	**-O0**	**-O1**	**-O2**	**-O3**	**-Ofast**	Notes
alias	any	any	any	any	typed	
div_split	off	off	off	off	on	on if IEEE_a=3
fast_complex	off	off	off	off	off	on if roundoff=3
fast_exp	off	off	off	on	on	on if roundoff>=1
fast_nint	off	off	off	off	off	on if roundoff=3
fast_sqrt	off	off	off	off	off	
fast_trunc	off	off	off	on	on	on if roundoff>=1
fold_reassociate	off	off	off	off	on	on if roundoff>=2
fold_unsafe_relops	on	on	on	on	on	
fold_unsigned_relops	off	off	off	off	off	
IEEE_arithmetic	1	1	1	2	2	
IEEE_NaN_inf	off	off	off	off	off	
recip	off	off	off	off	on	on if roundoff>=2
roundoff	0	0	0	1	2	
rsqrt	off	off	off	off	off	

For example, if you use `-OPT:IEEE_arithmetic` at `-O3`, the flag is set to `IEEE_arithmetic=2` by default.

7.8 Opteron performance

Although the Opteron platform has excellent performance, there are a number of subtleties in configuring your hardware and software that can each cause substantial performance degradations. Many of these are not obvious, but they can reduce performance by 30% or more at a time. We have collected a set of techniques for obtaining best performance described below.

7.8.1 Hardware setup

There is no "catch all" memory configuration that works best across all systems. We have seen instances where the number, type, and placement of memory modules on a motherboard can each affect the memory latency and bandwidth that you can achieve.

Most motherboard manuals have tables that document the effects of memory placement in different slots. We recommend that you read the table for your motherboard, and experiment.

If you fail to set up your memory correctly, this can account for up to a factor-of-two difference in memory performance. In extreme cases, this can even affect system stability.

7.8.2 BIOS setup

Some BIOSes allow you to change your motherboard's memory interleaving options. Depending on your configuration, this may have an effect on performance.

For a discussion of memory interleaving across nodes, see Section 7.8.3 below.

7.8.3 Multiprocessor memory

Traditional small multiprocessor (MP) systems use symmetric multiprocessing (SMP), in which the latency and bandwidth of memory is the same for all CPUs.

This is not the case on Opteron multiprocessor systems, which provide non-uniform memory access, known as NUMA. On Opteron MP systems, each CPU has its own direct-attached memory. Although every CPU can access the memory of all others, memory that is physically closest has both the lowest latency and highest bandwidth. The larger the number of CPUs, the higher will be the latency and the lower the bandwidth between the two CPUs that are physically furthest apart.

Most multiprocessor BIOSes allow you to turn on or off the interleaving of memory across nodes. Memory interleaving across nodes masks the NUMA variation in behavior, but it imposes uniformly lower performance. We recommend that you turn node interleaving off.

7.8.4 Kernel and system effects

To achieve best performance on a NUMA system, a process or thread and as much as possible of the memory that it uses must be allocated to the same single CPU. The Linux kernel has historically had no support for setting the affinity of a process in this way.

Running a non-NUMA kernel on a NUMA system can result in changes in performance while a program is running, and non-reproducibility of performance across runs. This occurs because the kernel will schedule a process to run on whatever CPU is free without regard to where the process's memory is allocated.

Recent kernels have some degree of NUMA support. They will attempt to allocate memory local to the CPU where a process is running, but they still may not prevent that process from later being run on a different CPU after it has allocated memory. Current NUMA-aware kernels do not migrate memory across NUMA nodes, so if a process moves relative to its memory, its performance will suffer in unpredictable ways.

Note that not all vendors ship NUMA-aware kernels or C libraries that can interface to them. If you are unsure of whether your kernel supports NUMA, check with your distribution vendor.

7.8.5 Tools and APIs

Recent Linux distributions include tools and APIs that allow you to bind a thread or process to run on a specific CPU. This provides an effective workaround for the problem of the kernel moving a process away from its memory.

Your Linux distribution may come with a package called `schedutils`, which includes a program called `taskset`. You can use `taskset` to specify that a program must run on one particular CPU.

For low-level programming, this facility is provided by the `sched_setaffinity(2)` call in the C library. You will need a recent C library to be able to use this call.

On systems that lack NUMA support in the kernel, and on runs that do not set process affinity before they start, we have seen variations in performance of 30% or more between individual runs.

7.8.6 Testing memory latency and bandwidth

To test your memory latency and bandwidth, we recommend two tools.

For memory latency, the LMbench package provides a tool called `lat_mem_rd`. This provides a cryptic, but fairly accurate, view of your memory hierarchy latency.

LMbench is available from `http://www.bitmover.com/lmbench/`

For measuring memory bandwidth, the STREAM benchmark is a useful tool. Compiling either the Fortran or C version of the benchmark with the following command lines will provide excellent performance:

```
$ pathf90 -Ofast stream_d.f second_wall.c -DUNDERSCORE
$ pathcc -Ofast -lm stream_d.c second_wall.c
```

(If you do not compile with at least `-O3`, performance may drop by 40% or more.)

The STREAM benchmark is available from `http://www.streambench.org/`

For both of these tools, we recommend that you perform a number of identical runs and average your results, as we have observed variations of more than 10% between runs.

Chapter 8

Examples

8.1 Compiler flag tuning and profiling with `pathprof`

We'll use the `168.wupwise` program from the CPU2000 floating point suite for this example. This is a Physics/Quantum Chromodynamics (QCD) code. For those who care, "wupwise" is an acronym for "Wuppertal Wilson Fermion Solver," a program in the area of lattice gauge theory (quantum chromodynamics). The code is in about 2100 lines of Fortran 77 in 23 files. We'll be running and tuning `wupwise` performance on the reference (largest) dataset. Each run takes about two to four minutes on a 2 GHz Opteron system to complete.

Even though this is a Fortran 77 code, the PathScale EKO Fortran compiler can handle it.

Outline:

Try `pathf90 -O2` and `pathf90 -O3` first.

Run times (user time) were:

```
          seconds
-O2        150.3
-O3        174.3
```

We're a little surprised since `-O3` is supposed to be faster than `-O2` in general. But the man page did say that the `-O3` "may include optimizations that are generally beneficial but may hurt performance."

So, let's look at a profile of the `-O2` binary. We do need to recompile using flags `-O2 -pg`.

Then we need to run the generated, instrumented binary again with the same reference dataset: `$ time -p ./wupwise > wupwise.out` (Here we used the `-p` (POSIX) flag to get a different time output format). This run generates the file `gmon.out` of profiling information.

Then we need to run `pathprof` to generate the human-readable profile.

```
$ pathprof ./wupwise
Flat profile:
Each sample counts as 0.01 seconds.
 %    cumulative  self              self     total
time   seconds   seconds  calls  s/call s/call name
51.15     83.54    83.54 155648000     0.00     0.00  zgemm_
17.65    112.37    28.83 603648604     0.00     0.00  zaxpy_
 8.72    126.61    14.24 214528306     0.00     0.00  zcopy_
 8.03    139.72    13.11 933888000     0.00     0.00  lsame_
 4.59    147.21     7.49                            s_cmp
 1.51    149.67     2.46   512301     0.00     0.00  zdotc_
 1.49    152.11     2.44 603648604     0.00     0.00  dcabs1_
 1.37    154.34     2.23 155648000     0.00     0.00  gammul_
 1.08    156.10     1.76 155648000     0.00     0.00  su3mul_
 1.07    157.85     1.75      152     0.01     0.50  muldeo_
...
 0.00    163.32     0.00        1     0.00   155.83  MAIN__
 0.00    163.32     0.00        1     0.00     0.00  init_
 0.00    163.32     0.00        1     0.00     0.06  phinit_
 %     the percentage of the total running time of the
time   program used by this function.
cumulative   a running sum of the number of seconds accounted
seconds      for by this function and those listed above it.
...
```

NOTE: The `pathprof` program is the complimentary version of `gprof` included in the PathScale EKO Compiler Suite.

Now, we note that the total time that `pathprof` measures is 163.3 secs. vs. the 150.3 that we measured for the original `-O2` binary. But considering that the `-O2 -pg` instrumented binary took 247 seconds to run, this is a pretty good estimate.

It is nice that the top hot-spot, `zgemm` consumes about 50% of the total time. We also note that some very small routines `zaxpy`, `zcopy`, and `lsame` are called a very large number of times. These look like ideal candidates for inlining.

In the second part of the `pathprof` output (after the explanation of the column headings for the flat profile), is a call-graph profile. In the example of such a profile below, one can follow the chain of calls from `main` to `matmul_`, `muldoe_`, `su3mul_`, and `zgemm_`, where most of the time is consumed.

```
================================================
Additional call-graph profile info:
                    Call graph (explanation follows)
   granularity: each sample hit covers 4 byte(s) for 0.01%
of 163.32 seconds
index % time    self  children    called     name
                0.00  155.83       1/1          main [2]
[1]     95.4    0.00  155.83       1          MAIN__ [1]
                0.00  151.19     152/152          matmul_ [3]
                0.05    4.47       1/1            uinith_ [13]
                0.00    0.06       1/1            phinit_ [22
                0.02    0.04       1/2            rndphi_ [21]
                0.00    0.00     301/512301       zdotc_ [14]
                0.00    0.00      77/1024077      dznrm2_ [17]
                0.00    0.00     452/603648604      zaxpy_ [9]
                0.00    0.00     154/214528306      zcopy_ [10]
                0.00    0.00      75/39936075      zscal_ [16]
                0.00    0.00       1/1            init_ [23]
-----------------------------------------------
  ...
 -----------------------------------------------
                0.00  151.19     152/152          MAIN__ [1]
[3]     92.6    0.00  151.19     152          matmul_ [3]
                1.75   73.84     152/152          muldoe_ [7]
                1.75   73.84     152/152          muldeo_ [6]
                0.00    0.00     152/214528306      zcopy_ [10]
                0.00    0.00     152/603648604      zaxpy_ [9]
-----------------------------------------------
                0.88   48.33 77824000/155648000     muldeo_ [6]
                0.88   48.33 77824000/155648000     muldoe_ [7]
[4]     60.3    1.76   96.65 155648000             su3mul_ [4]
               83.54   13.11 155648000/155648000    zgemm_ [5]
-----------------------------------------------
               83.54   13.11 155648000/155648000    su3mul_ [4]
[5]     59.2   83.54   13.11 155648000              zgemm_ [5]
               13.11    0.00 933888000/933888000    lsame_ [11]
-----------------------------------------------
  ...
================================================================
```

The `-ipa` option can analyze the code to make smart decisions on when and which routines to inline so we try that. `-O2 -ipa` results in a 133.8 second run time–a nice improvement over our previous best of 150 seconds with only `-O2`.

Since we heard somewhere that improvements with compiler flags are not always predictable, we also try `-O3 -ipa`. To our great surprise, we achieve a run time of 110.5 seconds, a 58% speed-up over our previous `-O3` time, and a nice speed-up over `-O2 -ipa`.

Section 7.7 mentions the flags `-O3 -ipa -LNO:fusion=2` and `-OPT:div_split=on`. Testing combinations of these two flags as additions to the `-O3 -ipa` we have already tested results in:

`-O3 -ipa -LNO:fusion=2` results in 109.74 seconds run time
`-O3 -ipa -OPT:div_split=on` results in 112.24 seconds
`-O3 -ipa -OPT:div_split=on -LNO:fusion=2` results in 111.28 seconds

So, `-O3 -ipa` is essentially a tie for the best set of flags with `-O3 -ipa -LNO:fusion=2`.

Chapter 9

Debugging and troubleshooting

9.1 Subscription Manager problems

For recommendations in addressing problems or issues with subscriptions, refer to Section 6.2 “Subscription problems" in the *PathScale EKO Compiler Suite Install Guide*.

9.2 Debugging

The earlier chapters on the PathScale EKO Fortran and C /C++ compilers contain language-specific debugging information. See Section 3.8 and Section 4.3. More general information on debugging can be found in this section.

The flag ’`-g`’ tells the PathScale EKO compilers to produce data in the form used by modern debuggers, such as GDB. This format is known as DWARF 2.0 and is incorporated directly into the object files. Code that has been compiled using `-g` will be capable of being debugged using GDB or other debuggers.

The `-g` flag does not affect the optimization level, but it is advisable to use `-O0` when debugging for most accuracy. If you use `-g`, you automatically use `-O0` optimization. Otherwise debugging may give unpredictable results.

9.3 Large object support

Statically allocated data (`.bss`) objects such as Fortran COMMON blocks and C variables with file scope are currently limited to 2GB in size. If the total size exceeds

that, the compilation (without the `-mcmodel=medium` option) will likely fail with the message:

```
relocation truncated to fit: R_X86_64_PC32
```

For Fortran programs with only one COMMON block or with no COMMON blocks after the one that exceeds the 2GB limit, the program may compile and run correctly.

At higher optimization levels (`-O3`, `-Ofast`), `-OPT:reorg_common` is set to `ON` by default. This might split a COMMON block such that a block begins beyond the 2GB boundary. If a program builds correctly at `-O2` or below but fails at `-O3` or `-Ofast`, try adding `-OPT:reorg_common=OFF` to the flags. Alternatively, using the `-mcmodel=medium` option will allow this optimization.

9.4 Using `-ipa` and `-Ofast`

When compiling with `-ipa,` the `.o` files that are created are not a regular `.o` files. IPA uses the `.o` files in its analysis of your program, and then does a second compilation using that information.

NOTE: When you are using `-ipa`, *all* the `.o` files have to have been compiled *with* `-ipa`, and *all* libraries have to have been compiled *without* `-ipa` for your compilation to be successful. In particular, when you link, all `*.o` files must have been compiled with `-ipa`, and all library archives (`libfoo.a`) must have been compiled without `-ipa`.

The requirement of `-ipa` may mean modifying Makefiles. If your Makefiles build libraries, and you wish this code to be built with `-ipa`, you will need to split these libraries into separate `*.o` files before linking.

By default, `-ipa` is turned on when you use `-Ofast`, so the caveats above apply to using `-Ofast` as well.

9.5 Tuning

Our compilers often optimize loops by eliminating the loop variable, and instead using a quantity related to the loop variable, called an "induction variable". If the induction variable overflows, the loop test will be incorrectly evaluated. This is a very rare circumstance. To see if this is causing your code to fail under optimization, try:

```
-OPT:wrap_around_unsafe_opt=off
```

Appendix A

Environment variables

This appendix lists environment variables utilized by the compiler, along with a short description. These variables are organized by language, with a separate section for those which are language independent.

A.1 Environment variables for use with C

PSC_CFLAGS - only passes flags to the the C compiler, `pathcc`

A.2 Environment variables for use with C++

PSC_CXXFLAGS - only passes flags to the C++ compiler, `pathCC`

A.3 Environment variables for use with Fortran

NLSPATH - flags for run-time and compile-time messages

F90_BOUNDS_CHECK_ABORT - set to `YES`, causes the program to abort on the first bounds check violation

PSC_FFLAGS - only passes flags to the Fortran compiler, `pathf90`

PSC_STACK_LIMIT - controls the stack size limit the Fortran runtime attempts to use

PSC_STACK_VERBOSE - Fortran runtime output about what it is doing with the stack size limit

A.4 Language independent environment variables

PSC_GENFLAGS - generic flags are passed to all compilers

Appendix B

Supported intrinsics

The following instrinsics are supported by the PathScale EKO Compiler Suite.

```
ABS(A)
ACOS(X)
ACOSD(X)
ADD_AND_FETCH(I, J)
ADJUSTL(STRING)
ADJUSTR(STRING)
AIMAG(Z)
AINT(A,KIND)
ALL(MASK, DIM)
ALLOCATED(ARRAY)
AND(I, J)
AND_AND_FETCH(I, J)
ANINT(A, KIND)
ANY(MASK, DIM)
ASIN(X)
ASIND(X)
ASSOCIATED(POINTER, TARGET)
ATAN(X)
ATAN2(Y, X)
ATAN2D(Y, X)
ATAND(X)
BITEST(I, POS)
BIT_SIZE(I)
BJTEST(I, POS)
BKTEST(I, POS)
BTEST(I, POS)
CCOS(X)
CDCOS(X)
CDEXP(X)
CDLOG(X)
CDSIN(X)
CDSQRT(X)
```

CEILING(A)
CEXP(X)
CHAR(I, KIND) intrinsic or ACHAR(I)
CLEAR_IEEE_EXCEPTION(EXCEPTION)
CLOC(C)
CLOCK()
CMPLX(X, Y, KIND)
COMPARE_AND_SWAP(I, J, K)
COMPL(I)
CONJG(Z)
COS(X)
COSD(X)
COSH(X)
COT(X)
COUNT(MASK, DIM)
CQCOS(X)
CQEXP(X)
CQLOG(X)
CQSIN(X)
CQSQRT(X)
CSHIFT(ARRAY, SHIFT, DIM)
CSIN(X)
CSMG(I, J, K)
CSQRT(X)
CVMGM(I, J, K)
CVMGN(I, J, K)
CVMGP(I, J, K)
CVMGT(I, J, K)
CVMGZ(I, J, K)
C_LOC(X)
DACOS(X)
DACOSD(X)
DASIN(X)
DASIND(X)
DATAN(X)
DATAND(X)
DATE()
DBLE(A)
DBLEQ(A)
DCOS(X)
DCOSD(X)
DCOSH(X)
DCOT(X)
DDIM(X,Y)
DEXP(X)
DFLOAT(A)
DFLOATI(A)
DFLOATJ(A)
DFLOATK(A)
DIGITS(X)
DIM(X,Y)
DISABLE_IEEE_INTERRUPT(INTERRUPT)
DLOG(X)
DLOG10(X)

DOT_PRODUCT(VECTOR_A, VECTOR_B)

DPROD(X, Y)

DREAL(A)

DSHIFTL(I, J, K)

DSHIFTR(I, J, K)

DSIGN(A, B)

DSIN(X)

DSIND(X)

DSINH(X)

DSM_CHUNKSIZE()

DSM_DISTRIBUTION_BLOCK()

DSM_DISTRIBUTION_CYCLIC()

DSM_DISTRIBUTION_STAR()

DSM_ISDISTRIBUTED()

DSM_ISRESHAPED()

DSM_NUMCHUNKS()

DSM_NUMTHREADS()

DSM_REM_CHUNKSIZE()

DSM_THIS_CHUNKSIZE()

DSM_THIS_STARTINGINDEX()

DSM_THIS_THREADNUM()

DSQRT(X)

DTAN(X)

DTAND(X)

DTANH(X)

DTIME()

ENABLE_IEEE_INTERRUPT(INTERRUPT)

EOSHIFT(ARRAY, SHIFT, BOUNDARY, DIM)

EPSILON(X) EQV(I, J)

EXIT(STATUS)

EXP(X)

EXPONENT(X)

FCD(I, J)

FDATE()

FETCH_AND_ADD(I, J)

FETCH_AND_AND(I, J)

FETCH_AND_NAND(I, J)

FETCH_AND_OR(I, J)

FETCH_AND_SUB(I, J)

FETCH_AND_XOR(I, J)

FLOATI(A)

FLOATJ(A)

FLOATK(A)

FLOOR(A)

FNUM

FP_CLASS(X)

FRACTION(X)

FREE(P)

FSTAT

GETPOS(I)

GET_IEEE_EXCEPTIONS(STATUS)

GET_IEEE_INTERRUPTS(STATUS)

GET_IEEE_ROUNDING_MODE(STATUS)

GET_IEEE_STATUS(STATUS)

HUGE(X)

IAND(I, J)

IBCHNG(I, POS)

IBCLR(I, POS)

IBITS(I, POS, LEN)

IBSET(I, POS)

ICHAR(C) intrinsic or IACHAR(C)

IDATE(I, J, K)

IDINT(A)

IEEE_BINARY_SCALE(Y, N)

IEEE_CLASS(X)

IEEE_COPY_SIGN(X, Y)

IEEE_EXPONENT(X, Y)

IEEE_FINITE(X)

IEEE_INT(X, Y)

IEEE_IS_NAN(X)

IEEE_NEXT_AFTER(X, Y)

IEEE_REAL(X, Y)

IEEE_REMAINDER(X, Y)

IEEE_UNORDERED(X, Y)

IEOR(I, J)

IFIX(A)

IIAND(I, J)

IIBCHNG(I, POS)

IIBCLR(I, POS)

IIBITS(I, POS, LEN)

IIBSET(I, POS)

IIDINT(A)

IIEOR(I, J)

IIFIX(A)

IINT(A)

IIOR(I, J)

IIQINT(A)

IISIGN(A, B)

ILEN(I)

IMVBITS(FROM, FROMPOS, LEN, TO, TOPOS)

INDEX(STRING, SUBSTRING, BACK)

ININT(A)

INOT(I)

INT(A, KIND)

INT1(A)

INT2(A)

INT4(A)

INT8(A)

INT_MULT_UPPER(I, J)

IOR(I, J)

IQINT(A)

IRTC()

ISHA(I, SHIFT)

ISHC(I, SHIFT)

ISHFT(I, SHIFT)

ISHFTC(I, SHIFT, SIZE)

ISHL(I, SHIFT)

ISIGN(A, B)

ISNAN(X)

JDATE()

JIAND(I, J)

JIBCHNG(I, POS)

JIBCLR(I, POS)

JIBITS(I, POS, LEN)

JIBSET(I, POS)

JIDINT(A)

JIEOR(I, J)

JIFIX(A)

JINT(A)

JIOR(I, J)

JIQINT(A)

JISIGN(A, B)

JMVBITS(FROM, FROMPOS, LEN, TO, TOPOS)

JNINT(A)

JNOT(I)

KIAND(I, J)

KIBCHNG(I, POS)

KIBCLR(I, POS)

KIBITS(I, POS, LEN)

KIBSET(I, POS)

KIDINT(A)

KIEOR(I, J)

KIFIX(A)

KILL

KIND(X)

KINT(A)

KIOR(I, J)

KIQINT(A)

KISIGN(A, B)

KMVBITS(FROM, FROMPOS, LEN, TO, TOPOS)

KNINT(A)

KNOT(I)

LBOUND(ARRAY, DIM)

LEADZ(I)

LEN(STRING)

LENGTH(I)

LEN_TRIM(STRING)

LGE(STRING_A, STRING_B)

LGT(STRING_A, STRING_B)

LLE(STRING_A, STRING_B)

LLT(STRING_A, STRING_B)

LOC(I) LOCK_RELEASE(I)

LOCK_TEST_AND_SET(I, J)

LOG(X)

LOG10(X)

LOG2_IMAGES()

LOGICAL(L, KIND)

LONG(A)

LSHIFT(I, POSITIVE_SHIFT)

M@CLR()

M@LD(X1)

M@LDMX(X1, X2)

M@MX(X1, X2)

M@UL()

MALLOC(P)

MASK(I)

MATMUL(MATRIX_A, MATRIX_B)

MAX(A1, A2, ... A63)

MAXEXPONENT(X)

MAXLOC(ARRAY, DIM, MASK)

MAXVAL(ARRAY, DIM, MASK)

MEMORY_BARRIER()

MERGE(TSOURCE, FSOURCE, MASK)

MIN(A1, A2, ... A63)

MINEXPONENT(X)

MINLOC(ARRAY, DIM, MASK)

MINVAL(ARRAY, DIM, MASK)

MOD(A, P)

MODULO(A, P)

MVBITS(FROM, FROMPOS, LEN, TO, TOPOS)

NAND_AND_FETCH(I, J)

NEAREST(X,S)

NEQV(I, J)

NINT(A, KIND)

NOT(I)

NULL(MOLD)

NUMARG()

NUM_IMAGES()

OMP_GET_DYNAMIC()

OMP_GET_MAX_THREADS()

OMP_GET_NESTED()

OMP_GET_NUM_PROCS()

OMP_GET_NUM_THREADS()

OMP_GET_NUM_THREADS()

OMP_GET_THREAD_NUM()

OMP_IN_PARALLEL()

OMP_SET_LOCK(LOCK)

OMP_TEST_LOCK(LOCK)

OMP_UNSET_LOCK(LOCK)

OR(I, J)

OR_AND_FETCH(I, J)

PACK(ARRAY, MASK, VECTOR)

POPCNT(I)

POPPAR(I)

PRECISION(X)

PRESENT(A)

PRODUCT(ARRAY, DIM, MASK)

QACOS(X)

QACOSD(X)

QASIN(X)

QASIND(X)

QATAN(X)

QATAND(X)

QCOS(X)

QCOSD(X)

QCOSH(X)

QCOT(X)

QDIM(X,Y)

QEXP(X)

QEXT(A)

QFLOAT(A)

QFLOATI(A)

QFLOATJ(A)

QFLOATK(A)

QLOG(X)

QLOG10(X)

QREAL(A)

QSIGN(A, B)

QSIN(X)

QSIND(X)

QSINH(X)

QSQRT(X)

QTAN(X)

QTAND(X)

QTANH(X)

RADIX(X)

RANDOM_NUMBER(HARVEST)

RANDOM_SEED(SIZE, PUT, GET)

RANF() RANGE(X)

RANGET(I)

RANSET(I)

READ@SM()

REAL(A, KIND)

REMOTE_WRITE_BARRIER()

REM_IMAGES()

REPEAT(STRING, NCOPIES)

RESHAPE(SOURCE, SHAPE, PAD, ORDER)

RRSPACING(X)

RSHIFT(I, NEGATIVE_SHIFT)

RTC() SCALE(X,I)

SCAN(STRING, SET, BACK)

SELECTED_INT_KIND(R)

SELECTED_REAL_KIND(P,R)

SET_EXPONENT(X,I)

SET_IEEE_EXCEPTION(EXCEPTION)

SET_IEEE_EXCEPTIONS(STATUS)

SET_IEEE_INTERRUPTS(STATUS)

SET_IEEE_ROUNDING_MODE(STATUS)

SET_IEEE_STATUS(STATUS)

SHAPE(SOURCE)

SHIFT(I, J)

SHIFTA(I, J)

SHIFTL(I, J)

SHIFTR(I, J)

SHORT(A)

SIGN(A, B)

SIGNAL()

SIN(X)

SIND(X)

SINH(X)

SIZE(ARRAY, DIM)

SIZEOF(X)

SNGL(A)

SNGLQ(A)

SPACING(X)

SPREAD(SOURCE, DIM, NCOPIES)

SQRT(X)

STAT

SUB_AND_FETCH(I, J)

SUM(ARRAY, DIM, MASK)

SYNCHRONIZE()

SYNC_IMAGES([image])

SYSTEM_CLOCK(COUNT, COUNT_RATE, COUNT_MAX)

TAN(X)

TAND(X)

TANH(X)

TEST_IEEE_EXCEPTION(EXCEPTION)

TEST_IEEE_INTERRUPT(INTERRUPT)

THIS_IMAGE([array[,dim]])

TIME(BUF)

TIME8

TINY(X)

TRANSFER(SOURCE, MOLD, SIZE)

TRANSPOSE(MATRIX)

TRIM(STRING)

UBOUND(ARRAY, DIM)

UNIT(I)

UNPACK(VECTOR, MASK, FIELD)

VERIFY(STRING, SET, BACK)

WRITE_MEMORY_BARRIER()

XOR(I, J)

XOR_AND_FETCH(I, J)

ZABS

ZCOS

ZEXP

ZLOG

ZSIN

ZSQRT

Appendix C

Glossary

The following is a list of terms used in connection with the PathScale EKO Compiler Suite.

AMD64 AMD's 64-bit extensions to Intel's IA32 (more commonly known as "x86") architecture. The AMD64 extensions are referred to by Intel as IA32e.

alias An alternate name used for identification, such as for naming a field or a file.

aliasing Two variables are said to be "aliased" if they potentially are in the same location in memory. This inhibits optimization. A common example in the C language is two pointers; if the compiler cannot prove that they point to different locations, a write through one of the pointers will cause the compiler to believe that the second pointer's target has changed.

assertion A statement in a program that a certain condition is expected to be true at this point. If it is not true when the program runs, execution stops with an output of where the program stopped and what the assertion was that failed.

base Set of standard flags used in SPEC runs with compiler.

bind To link subroutines in a program. Applications are often built with the help of many standard routines or object classes from a library, and large programs may be built as several program modules. Binding links all the pieces together. Symbolic tags are used by the programmer in the program to interface to the routine. At binding time, the tags are converted into actual memory addresses or disk locations. Or (bind) to link any element, tag, identifier or mnemonic with another so that the two are associated in some manner. See *alias* and *linker*.

CG Code generation; a pass in the PathScale EKO Compiler.

common block A Fortran term for variables shared between compilation units (source files). Common blocks are a Fortran-77 language feature that creates a group of global variables. The PathScale EKO compiler does sophisticated padding of common blocks for higher performance when the Inter-Procedural Analysis (IPA) is in use.

constant A constant is a variable with a value known at compile time.

DSO (dynamic shared object) A library that is linked in at runtime. In Linux, the C library (glibc) is commonly dynamically linked in. In Windows, such libraries are called DLLs.

DWARF A debugging file format used by many compilers and debuggers to support source level debugging. It is architecture-independent and applicable to any processor or operating system. It is widely used on Unix, Linux, and other operating systems, as well in stand-alone environments.

EBO The Extended Block Optimization pass in the PathScale EKO compiler.

equivalence A Fortran feature similar to a C/C++ union, in which several variables occupy the same are of memory.

feedback A compiler optimization technique in which information from a run of the program is then used by the compiler to generate better code. The PathScale EKO Compiler Suite uses feedback information for branches, loop counts, calls, switch statements, and variable values.

flag A command line option for the compiler, usually an option relating to code optimization.

gcov A utility used to determine if a test suite exercises all code paths in a program.

IPA (Inter-Procedural Analysis) A sophisticated compiler technique in which multiple functions and subroutines are optimized together.

linker A utility program that links a compiled or assembled program to a particular environment. Also known as a "link editor," the linker unites references between program modules and libraries of subroutines. Its output is a load module, which is executable code ready to run in the computer.

LNO (loop nest optimizer) Performs transformation on a loop nest, improves data cache performance, improves optimization opportunities in later phases of compiling, vectorizes loops by calling vector intrinsics, parallelizes loops, computes data dependency information for use by code generator, can generate listing of transformed code in source form.

MP Multiprocessor

NUMA Non-uniform memory access is a method of configuring a cluster of microprocessors in a multiprocessing system so that they can share memory locally, improving performance and the ability of the system to be expanded. NUMA is used in a symmetric multiprocessing (SMP) system.

pathcov The version of `gcov` that PathScale supports with its compilers. Other versions of `gcov` may not work with code generated by the PathScale EKO Compiler Suite, and are not supported by PathScale.

pathprof The version of `gprof` that PathScale supports with its compilers. Other versions of `gprof` may not work with code generated by the PathScale EKO Compiler Suite, and are not supported by PathScale.

peak Set of optional flags used with compiler in SPEC runs to optimize performance.

SIMD (Single Instruction Multiple Data) An i386/AMD64 instruction set extension which allows the CPU to operate on multiple pieces of data contained in a single, wide register. These extensions were in three parts, named MMX, SSE, and SSE2.

SMP Symmetric multiprocessing is a "tightly-coupled," "share everything" system in which multiple processors working under a single operating system access each other's memory over a common bus or "interconnect" path.

SPEC (Standard Performance Evaluation Corporation) SPEC provides a standardized suite of source code based upon existing applications that has already been ported to a wide variety of platforms by its membership. The benchmarker takes this source code, compiles it for the system in question and tunes the system for the best results. See http://www.spec.org/ for more information.

TLB Translation Lookaside Buffer

vectorization An optimization technique that works on multiple pieces of data at once. For example, the PathScale EKO Compiler Suite will turn a loop computing the mathematical function `sin()` into a call to the `vsin()` function, which is twice as fast.

WHIRL The intermediate language (IR) used by compilers allowing the C, C++, and Fortran front-ends to share a common backend. It was developed at Silicon Graphics Inc. and is used by the Open64 compilers.

Index

-C, 20
-O0, 16, 43
-O1, 16, 43
-O2, 16, 43
-O3, 16, 44
-Ofast, 30, 66
-cpp, 7
-ff2c-abi, 25
-fixedform, 15
-fno-second-underscore, 24
-fno-underscoring, 24
-freeform, 15
-ftpp, 7, 16, 19
-g, 11, 32, 65
-ipa, 30, 39, 45, 66
-lm, 31
-pg, 12
-traditional, 19
-v, 6
.F, 15, 19
.F90, 15, 19
.f, 7, 15
.f90, 7, 15
.o files, 45

ABI, 5, 24
alias analysis, 52
aliasing, 52
AMD64, 5
apropos, 2, 6
autoconf, 35

basic optimization, 39, 43
BIOS setup, 58
bounds checking, 20

C compiler compatibility, 29
C preprocessor, 31
C++ compiler compatibility, 29
cache blocking, 49
cache size, 48
CMOVE, 56
code generation, 50
code tuning example, 61
COMMON block, 66
compat-gcc, 33
compat-gcc script, 34
compiler environment variables, 69
compiler options, 9
Compiler Quick Reference, 5
COMPLEX, 25
conventions, 2
Cray pointer, 18

default optimization level, 30
disable a feature, 45
DWARF, 11, 65

enable a feature, 45
environment variables for compat-gcc, 35

fbdata, 51
feedback-directed optimization, 40, 51
fixed-form, 15
Fortran compatibility, 15
Fortran compiler stack size, 36
Fortran KIND, 22
Fortran modules, 17
Fortran runtime libraries, 31
Fortran stack size, 16

g77, 25, 33
gcc, 33
GCC compilers, 30
GDB, 11
gmon.out, 12
gprof, 12, 41, 61, 62
group optimizations, 45

hardware setup, 57
higher optimization, 40

IEEE 754 compliance, 54
IEEE arithmetic, 55
induction variable, 67
inner loop unrolling, 49
input file types, 7
interleaving, 58

limit, 16
loop fission, 48
loop fusion, 48
loop fusion and fission, 47
Loop Nest Optimization (LNO), 47
loop unrolling, 49

man pages, 2
memory latency and bandwidth, 59
modifying Makefiles, 46
modifying scripts, 35
multiprocessor memory, 58

name mangling, 35
Non-Temporal at All (NTA), 50
NUMA, 58

object files from .f90 files, 8
Opteron performance, 57
outer loop unrolling, 49

pathCC, 30
pathcc, 30
pathf90, 15
peeling, 48
Prefetch, 47
prefetch, 49
PRNG, 20

REAL, 25
roundoff, 55

schedutils, 59
shared runtime libraries, 9
statically allocated data, 65
STREAM, 59

taskset, 59
tiling, 49
time, 41
time tool, 12
TLB, 47
Tuning Quick Reference, 39

ulimit, 16

vectorization, 50

www.pathscale.com, 2

PathScale™

PATHSCALE, INC.
477 NORTH MATHILDA AVENUE
SUNNYVALE, CA 94085 USA

TEL 408.746.9100
FAX 408.746.9150
PATHSCALE.COM